Diplomarbeit

An Ontology for Digital Forensics in IT Security Incidents

Jan Peter Wolf

Universität Augsburg

Fakultät für Angewandte Informatik

Institut für Informatik/Institut für Software & Systems Engineering

Gutachter:

Prof. Dr. Wolfgang Reif

Prof. Dr. Bernhard Bauer

An Ontology for Digital Forensics in IT Security Incidents

Jan Wolf

Acknowledgement

I would like to thank all those who participated in making this piece of work a reality. Special thanks go to Thomas Schreck who suggested the topic of this thesis and guided me through the field of forensics. Additionally I would like to express my gratitude towards Dr. Kurt Stenzel who made this work possible. Special thanks also to all further proof readers who gave me valuable feedback, namely Gabriele Binner and Christoph Lassner. Last but not least I would like to thank my family and my girlfriend for supporting me during all the time of work.

Contents

Chapter 1

Introduction

> "[The] [...] “Golden Age of Digital Forensics,” [...] is quickly coming to an end." [Garfinkel, 2010]
>
> Ontologie, die Lehre des Seienden.
> (von ὄν seiend, Partizip von εἶναι sein, und λόγο(ς) Lehre)

For a long time computers have been utilized for investigating criminal cases. Computer databases are used to find information faster than in large paper document stores. If evidence has to be reconstructed, computers for example help reconstruct ripped up documents[De Smet, 2009]. As seen in television series, computers can help identify footprints[Huynh et al., 2003].

But computers and other electronic devices can contain evidence or be evidence themselves, too. Guidelines for what evidence can be found in which electronic device, how the evidence can be retrieved and what precautions to take are available for example in [National Institute of Justice (U.S.), 2001].

Digital forensics has to face several difficulties. The solutions for these do not necessarily go in the same direction. On the one hand the available data should be processed completely. On the other hand this has to be done as fast as possible.

One factor that leads to problems is the lately increasing number of mobile devices. Many of them have different structures and require different approaches. An additional point is that the memory of such a device cannot easily be taken out as it is possible to remove the hard disk from a common computer. This leads to problems if the memory is used as legal evidence. Another aspect is the amount of space available for and used by users. Today they can have large storage built in their computer. Furthermore most of them have several external storage media.

All these points increase the complexity of retrieving the required information and the time needed for analysing it. Caused by the fact that the digital forensic analysis is based on traditional forensics, the first approach comprises that the data is acquired first and analysed later on. Caused by the rapidly growing amount of data, it is more efficient to first filter what

data leads to finding evidence and gathering specific additional information later on.

Another point that has changed is that much evidence can only be found on live systems. Caused by the fact that a live system runs on, some information might vanish which complicates the retrieval of additional facts.[Adelstein, 2006]

But the person who has to acquire a computer as evidence does not necessarily know that there is volatile data and how to seize it correctly. In [National Institute of Justice (U.S.), 2004] it is mentioned that evidence can be volatile but in the chapter about acquisition it is assumed that the computer is powered off prior to the examination.

Another problem is that the tools built for helping forensic analysts and investigators do not always work as they are supposed to. Some of them can be convinced to work together, but there are only few standards for exchanging data. Many of the tools are developed from scratch. This leads to the problem that similar program parts are developed multiple times whereas the used effort could be used more productively.[Garfinkel, 2010]

Caused by the fact that researchers develop new tools for the field they are proficient with, the resulting products are made only for this purpose. As a result there are plenty of tools an investigator has to be able to use for solving a single case. And the output of one tool needs to be adjusted to be compatible to another tool. An additional problem is that new techniques are developed and presented but without a fully working implementation[Tang and Daniels, 2005].

This work introduces an ontology for forensic analysis. By using the ontology the output of tools can be put to one place like in traditional databases but furthermore it allows to automatically draw conclusions about the correlation of the single results.

The topic for this work was issued by Siemens CERT Munich. At first an ontology that represents the forensically interesting parts of a computer had to be implemented. As next move an example implementation of a program had to be built that converts data provided by existing forensic tools to a format that matches the constraints of the ontology had to be built. Additionally queries had to be written that allow to find evidence in the converted data. Furthermore the functionality needed to be tested on real malware.

Chapter 2

Related Work

In [Garfinkel, 2010] it is outlined that digital forensics have grown important in recent time but it "largely lacks standardization and process" and that there has not yet been found a solution for "the lack of intelligent analytics beyond full-text search, non-standard computing devices (especially small devices), ease-of-use, and a laundry list of unmet technical challenges".

One of the problems concerning standardization is that the available data sources have different formats. For example there exist plenty of different formats for hard disks, called file systems, some with great other with less differences. A forensic analyst therefore needs to be familiar with the details of the most popular ones. This necessity can be treated by developing a superior classification of the data stored on the disk. The hard disks can be divided into the same categories, regardless of the format, as shown in [Carrier, 2005]. This model for file systems is not only useful for theoretical comparison but can be used practically. The author has written a collection of tools with one for each aspect of this model. Among other things this includes one tool for determining the disk layout, one for the list of files. A closer look at the categorization and the tools is taken in sections 4.2.1 and 4.3.1.

The same issue imposes with the random access memory. And it can be treated equally. The data can be categorized in a similar manner. Details on the model can be found in section 4.2.2 and one tool for retrieving the needed data is presented in section 4.3.2.

Caused by the increasing amount of data that has to be processed, efforts are being made to automate as much of the investigative work as possible. This can for instance be seen in [Garfinkel, 2009] where a tool is being presented that analyses a hard disk and generates an XML description of it. Prior to this tool the aforementioned set of utilities had to be used to generate an overview of the hard disk. But it takes much more time to run many programs by hand.

In [Farrell, 2009] a batch reporting system is introduced that allows the

investigator to get a fast overview of the available hard disk. It is based on *PyFlag*[Cohen, 2012] which uses *The Sleuth Kit*[Carrier, 2012a] to extract the data from the hard drive respectively the image of the hard drive.

An ontology and the associated data storage is preferred to a relational database concept because the relational database was not built for storing graph structures[Neo Technology, Inc., 2006].

Chapter 3

Goal Forensic Semantic Model

This chapter explains what this work is about. Additionally it introduces the example that will be used throughout this work.

3.1 Ontology

The goal is to create an ontology, a data store, a program to fill the data store, and several queries that allow the examiner to obtain conclusive evidence fast and easily. In the end, only the sources to obtain the data from have to be selected and the program extracts the necessary information and puts it into the appropriate structure. After the data was imported into the data store the examiner can query for evidence. If not marked otherwise the term "ontology" stands for the one that is introduced in this work.

It is intended to make the gathering of data and the extraction of evidence faster and more effectively. One positive side effect of the automatic importing is that there is no need to know every option of every tool involved, so even novice users can use the program to gather evidence and the forensic expert can concentrate on more difficult tasks. Another point is that multiple users can access the data store so numerous examiners can work together on one case.

The procedure is mainly intended for the fast examination of cases that incident response teams have to take care of. A requirement from the ontology is that it has to be customizable for special needs. This can be relevant if additional information is required that is not yet represented yet. An example for this is explained in section 4.2.3.3. The tools that provide the information collected in the data store have to be interchangeable, what is another important fact. This allows examiners to use the tools they are familiar with.

The idea is to build an ontology which represents the forensically interesting parts of a computer system. Chapter 4 will explain how to extract which part of information and why they are interesting. The information is

gathered from the unit of analysis with several of the available tools which are presented in section 4.3.

When all these pieces are put into the data store according to the structure given by the ontology, connections are re-established as they were on the computer system. For example, processes in the random access memory and corresponding open files on the hard disk get connected. The automatic creation of connections between different parts allows seamless transition between different data sources without the need to run different tools and to interpret the different output formats.

3.2 Example

This example explained here will be used throughout the rest of this work. It will only contain a very small amount of data to keep it easy. It would be possible to use real data but that would just increase the amount of data and bring no further advantage for the understanding of the concepts. Nevertheless the application of the ontological approach on real cases will be discussed in section 8 where real malware samples are analysed.

For the example we take a virtual computer with one hard disk, one network card, some random access memory and a *Microsoft Windows* like operating system.

We have not been careful enough and some malware has caused damage on the system. To keep the example small, the system contains only a minimal amount of data.

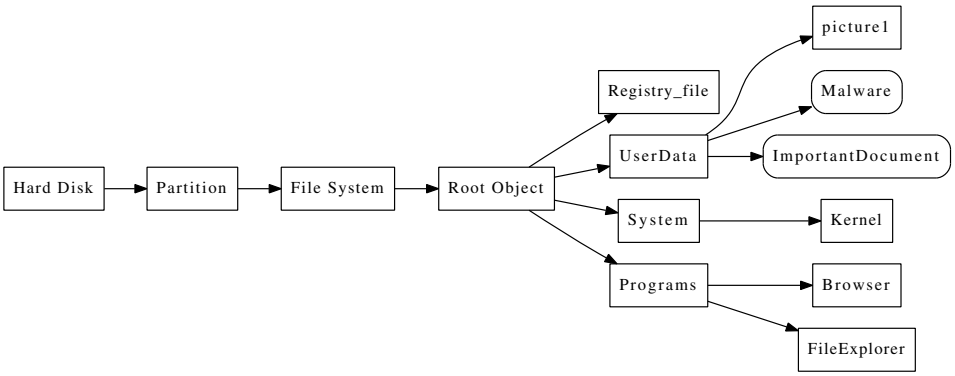

Figure 3.1: Hard Disk example

The hard disk has the content shown in figure 3.1. The squares represent files and the ones with rounded corners are deleted.

The file `Malware` is deleted because the malware wants to hide itself after it is started. The `ImportantDocument` file was deleted by the user. Whether this was done intentionally or by accident may be relevant for the case but not for the concept. In the case that the user has collected data he should

not possess, he deleted it to remove traces. On the other hand the malware might be intended to delete particular files.

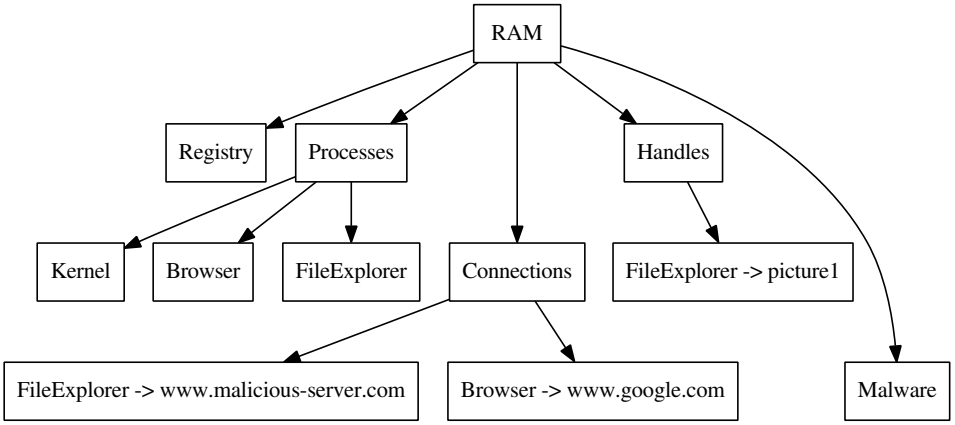

Figure 3.2: Random Access Memory example

The random access memory contains a copy of the `Registry`, the three processes `Kernel`, `Browser` and `FileExplorer`, handles and connections between processes and to network resources. Figure 3.2 roughly visualizes the data that can be found. The `Malware` node is not connected to the `Processes` node because it is not listed in the normal list of processes but as it is stored there, it is connected to the `RAM` node. More details of the registry and the information available there, are explained in section 4.2.3.1.

The `Registry` is a collection of key value pairs that is mainly used for the storage of configuration data. The information that can be found in the `Processes` section includes currently running processes, some processes that have already finished and processes that try to hide their existence. In the `Connections` category there are stored connections between programs and to network addresses that are currently open and some of them that are already closed. `Handles` include all other resources, for example files, that a process can access.

The content of the `Registry` file and the `Registry` memory object will be explained later.

The analysis of the data is continued in section 4.1.3.

Chapter 4

Forensics

This chapter will explain what data is needed for the forensic analysis and introduce tools and techniques for retrieving, storing and accessing the data.

4.1 Forensics Basics

Forensics, short for forensic science, means the systematic application of scientific methods for answering questions. The majority of the questions belong to the solution of criminal acts or are otherwise connected to them. Forensic methods are used in order to provide reasonable reproducible facts for the solution. As forensic methods are successfully applied on cases in various domains for a long time, they were adapted to the needs of cases that involve computers.[Kruse and Heiser, 2001]

The term malware is used as a general label for all kinds of malicious software. Among other things this includes viruses, rootkits, trojan horses, worms, and dialers.

4.1.1 Real Case

In the United States of America the famous case of the BTK-killer was solved by the help of computational forensics after several decades. The perpetrator killed ten people and sent letters to the police in the years of 1974 to 1991. In 2004 he sent a floppy disk with his last message. Forensic examiners found the decisive information in the metadata of the document file.[The Associated Press, 2012, IADT Chicago, 2011]

4.1.2 Cyber Forensics

In this document “forensics” specifically means computational forensics, also called IT-forensics or cyber forensics. This "involves preservation, identification, extraction, documentation, and interpretation of computer media for evidentiary and/or root cause analysis"[Kruse and Heiser, 2001].

Basic rules how to correctly examine cases are listed in [Kruse and Heiser, 2001]. The guidelines in [National Institute of Justice (U.S.), 2004] demand similar precautions including:

- One should acquire the evidence without altering or damaging the original data.
- It has always to be provable that the data where evidence is found is the same as the original one and has not been modified during analysis.
- Every step needs to be documented to prove a complete chain of custody.

Digital forensics has three phases according to [Carrier, 2003]:

1. Acquisition
2. Analysis
3. Presentation

In phase one the data is collected. In phase two this data is analysed to find pieces of evidence. The ontology belongs to the analysis section as it gathers the information collected in the acquisition phase and simplifies the detection of evidence. The third phase differs depending on where the evidence are used. The evidence need to be presented in another way if used in a case before court, in a corporate investigation or anywhere else.

In [National Institute of Justice (U.S.), 2001] electronic devices to look for data are presented and the kind of evidence that can be found there. Additionally caveats are mentioned that concern the retrieval or the storage of the data. Furthermore procedures for the handling of these devices are explained in order not to destroy or tamper evidence.

For investigating a case the examiner will first take a snapshot of the computer. This is useful as the original data can stay unchanged and makes it easier to prove the consistency to the snapshot if required in the course of the case. In most cases the snapshot contains hard disk and random access memory data. These snapshots are often called dumps, images, or samples. This can be compared to taking pictures and other samples of a crime scene.[Carrier, 2003]

If taking the snapshot from a running (live) system one problem is that it can hardly be granted that the snapshot is taken at a specific point in time. In particular it means that the data is possibly being changed while the snapshot is being taken. As the system is running, programs can change data. Additionally it takes time to take the snapshot and in this time the data already read and to be read might change so that the snapshot is not consistent. Furthermore, if a program writes to the memory where the tool for retrieving the snapshot reads, it can tamper with the integrity of the

data. Both of this can lead to (un)intentional destruction of traces.[Vidas, 2007]

According to [Kruse and Heiser, 2001, pp 5 f] many investigators recommend to pull the power plug to stop the computer and with it the malware. This freezes the hard disk as it is and the malware can no longer pursue its objective. Others recommend to take the snapshot from the live system because then also the volatile information can be used. Yet others even prefer a normal shut down process. Maybe a hybrid approach is acceptable that first captures the volatile information from the live system and then pulls the plug. But there is no best practice for all cases.

4.1.2.1 Hard Disk

The data from the hard disk can be obtained from a live system or using a hardware or software write block. A write block is used to prevent write operations. Using a write block eliminates or at least reduces the risk of changing data on the piece of evidence. The live system alternative is limited by the access rights the investigator has to the system and by the problems with live systems as mentioned before.

Both kinds of write blocks also have their limitations, hardware ones can be very expensive and require the disk to be removed from the computer. On the other hand software ones may be easier to use, as there is mostly no need to remove the disk from the computer, but they are not as reliable as hardware ones. An additional restriction for hardware blocks is that for removing the disk the computer has to be powered off, which can lead to losing volatile evidence.

To limit the loss of volatile evidence it should be acquired first. The question must be considered whether to pull the plug and risk damaged data or shut down properly and risk that the malware cleans up traces. One limitation also for both write block alternatives is that some data that is recorded by the hardware itself, such as S.M.A.R.T. data, is changed nevertheless. The "Self-Monitoring, Analysis and Reporting Technology"(S.M.A.R.T.) data is acquired by the hardware itself to help prevent data loss. It includes values that can be used to predict disk failure for example *temperature* and *power-on hours*. If the disk is powered on the *power-on hours* value increases regardless of any write block.

Another aspect when retrieving data from a hard disk is that it can have hardware damages. One does not know whether operating such a disk is safe or destroys more data. Most hard disks have a mechanism that can reestablish consistency for a number of damaged storage units by using spare units and error correction codes. But one does not know if the hardware destroys evidence in doing this.[Council and Institute, 1998][Ewert and Schultz, 1992]

4.1.2.2 Random Access Memory

The data from the random access memory can be obtained via software or via hardware.

The hardware solution can be expensive and is not assured to work on all computers. One solution for example requires a special expansion card to be installed in the computer prior to the incident[Carrier and Grand, 2004]. This may be a solution for servers but it is not really feasible for every computer. Another one relies on the firewire port which is not necessarily built in every computer. An further problem of this method is that some systems crash as a consequence of the acquisition. And if the system crashes, the volatile evidence is lost. The hardware based acquisition of the memory can also be compromised by special malware[Rutkowska, 2007]. This interfering of the malware is not necessarily removable without rebooting the system and losing the volatile evidence.

The software solution needs to write data to the computer in order to work. The data that is written is called footprint of the tool. This footprint could overwrite evidence and reduce the amount of available data for analysis. Depending on the method used it is smaller or larger, so the examiner has to be aware of the limitations of the involved tools. Limitations of some methods are provided in [Davis, 2008] and [Vidas, 2007].

But there is another uncertainty that can destroy evidence, namely the user. If he pulls the plug when he detects that something is wrong, the volatile memory is lost and with it all traces left there. Even if he lets the system run on, he can (un)intentionally cover tracks.

Talking about forensics and cyber crime most people consider that an evil minded person sits somewhere and writes malware to do evil things. But forensics are also applicable when for example employees that are unhappy about their employer want to gain some extra money by extracting sensitive data, maybe without having the right to obtain the data, and selling it. And for hiding his doing, tracks have to be covered intentionally. Examining cases about malware or industrial espionage are only two of many application fields of forensics.

After introducing how to retrieve the data from a computer the next chapter explains the structure of this data.

4.1.3 Example

To continue the example of section 3.2 it is assumed that the snapshots are taken from the virtual computers hard disk and random access memory when the system is paused. That way they are not changed during the process. In section 4.2.4 the data will be split up.

4.2 Data

The ontology is a model for a computer system and the data that was gathered needs to be structured with forensics in mind. Additionally, this structure must be applicable to most data formats because the ontology should be useful regardless of special data formats.

4.2.1 Hard Disk

A hard disk is basically a store for data. Because this store can contain a large collection of information structures called filesystem, formats were invented for structuring the data to be easier stored and found. There are many different formats for managing hard disks. The most common ones are FAT, NTFS, EXTx and HFS but there are a lot more. Each of them has their own structure and thus would need separate treatment. In [Carrier, 2005] the author introduces a categorization for the information commonly available in the file system formats that can be applied to most of them. This meta structure is used in the ontology. Depending on the file system type to analyse one only has to map it to this structure to use the ontology. Figure 4.1 shows the structure.

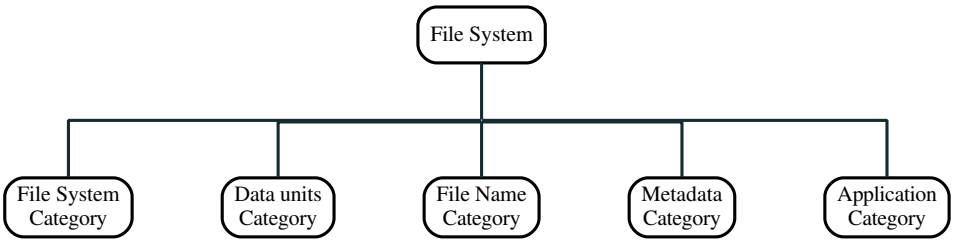

Figure 4.1: Hard disk structure

According to this categorization the data on a hard disk can be divided into the following categories:

- `File System`
  The basic information of the file system, like name, type, version number, and additional file system specific data. Additionally there is data structure that links to all currently available files.

- `Data units`
  The real data of the files on the disk. The sectors of the hardware are structured as the file system requires. For example a file system can unite three hardware sectors to one data unit.

- `File Name`
  There is at least one file name entry for each file in the file system. This

entry contains the name of a file, reference to corresponding metadata entries and a list of file name entries for all children.

- `Metadata`
  Similar to the file name entries there is at least one metadata entry for each file. This entry contains additional information for a file like access rights, ownership, access and creation times and additional flags. Furthermore it contains links to file name entries and data units.

- `Application specific`
  Some file system types allow special information like journals, quota restrictions, logs or other file system specific options.

How the example from section 3.1 splits up to these categories is shown later in section 4.3.1 that explains the corresponding tool.

4.2.2 Random Access Memory

Similar to the hard disk, the structure of the data in random access memory depends on the system it is managed by. A big difference to the hard disk is that the random access memory is volatile. This means, its content is lost when the power is taken away from it.

There is not yet a source for this as the model is developed within Siemens CERT and will be published. Like the hard disk, the memory structures can be mapped to a meta structure as shown in figure 4.2. [Schreck et al. Siemens CERT,]

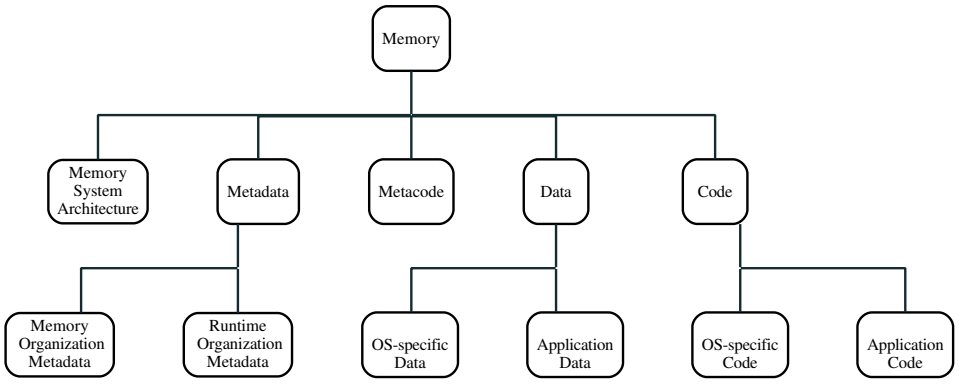

Figure 4.2: Random access memory structure

The categories contain the following information[Schreck et al. Siemens CERT,]:

- `Memory System Architecture`
  This category contains information that is necessary to boot a system.

It comprises for example physical memory layout information, boot code, setup code, and the size of the page tables.

- `Metadata`
  - `Memory Organization`
    This is the data that contains information about the organization of the memory. It is necessary to manage the memory and includes for example the page tables.
  - `Runtime Organization`
    This is the data that is necessary for the management of data within the memory. It comprises for example the process list and the IO management.
- `Metacode`
  This section contains the code necessary to operate the system and influence the behaviour of the operating system. This includes for example interrupt service routines, page fault handler, IO code, and scheduler.
- `Code`
  - `OS-specific`
    This is the code that is used by the operating system to implement OS-specific functionality. It comprises for example daemon processes and idle process.
  - `Application`
    This is the code that is used by user processes.
- `Data`
  - `OS-specific`
    This is the data that is used by the operating system processes.
  - `Application`
    This is the data that is used by the user processes.

4.2.3 Additional information

The data in this section differs from the already mentioned one as it can be derived from one or more of them.

4.2.3.1 Registry

Information about the registry data can be found in random access memory and on the hard disk but there is a difference. In the memory there are parts of the registry that are only necessary at runtime and can therefore not be found on the disk. Forensically interesting is the difference of the data from the two sources. One cause for differences is that changes in the registry are not always immediately written to disk respectively memory. Another one is that malware tries to manipulate the computer by changing one or both sources.

The naming of the different parts of the registry is inspired by the *Microsoft Windows* registry structure. Other configuration stores can also be mapped to this model even though the main target system is the *Microsoft Windows* family.

The registry is made up of hives. Hives are the different files that contain configuration information. One hive contains the data of both sources to make it easier to spot differences. The hives themselves have a tree structure, so every entry in the tree, called key, can have hive-values, the final configuration data, and sub-keys. The keys can have a state-flag that tells whether it can be found only in volatile memory or in both sources. The hive-values are tuples with key, value and data type of the value. An example hive is shown in figure 4.3.

As prior mentioned, the configuration of other systems can be mapped to this structure. For example the configuration of the *Gnome Desktop* is also structured as a tree[The GNOME Project, 2011].

4.2.3.2 Network

Similar to the registry data this data is acquired from one or more of the sources above. Interesting network information include current IP addresses and connections, gateways, and name server. If the malware for example wants to redirect the user to manipulated or forged websites it may change the name server as done by the DNSChanger[Federal Bureau of Investigation, 2011].

4.2.3.3 Other data on a computer

Of course there is more information stored on a computer. One example are log files. These can provide information about events on the system and when they occurred. From this perspective they are forensically interesting. As mentioned in [Kruse and Heiser, 2001, pp 291f] they are not necessarily trustworthy. At first logging has to be enabled and working prior to the incident. Then there is the question about authenticity of the log entries. Some malware can create and/or edit log entries and thus obfuscate or delete traces. If the log is for example a normal file on the computer, it is similarly

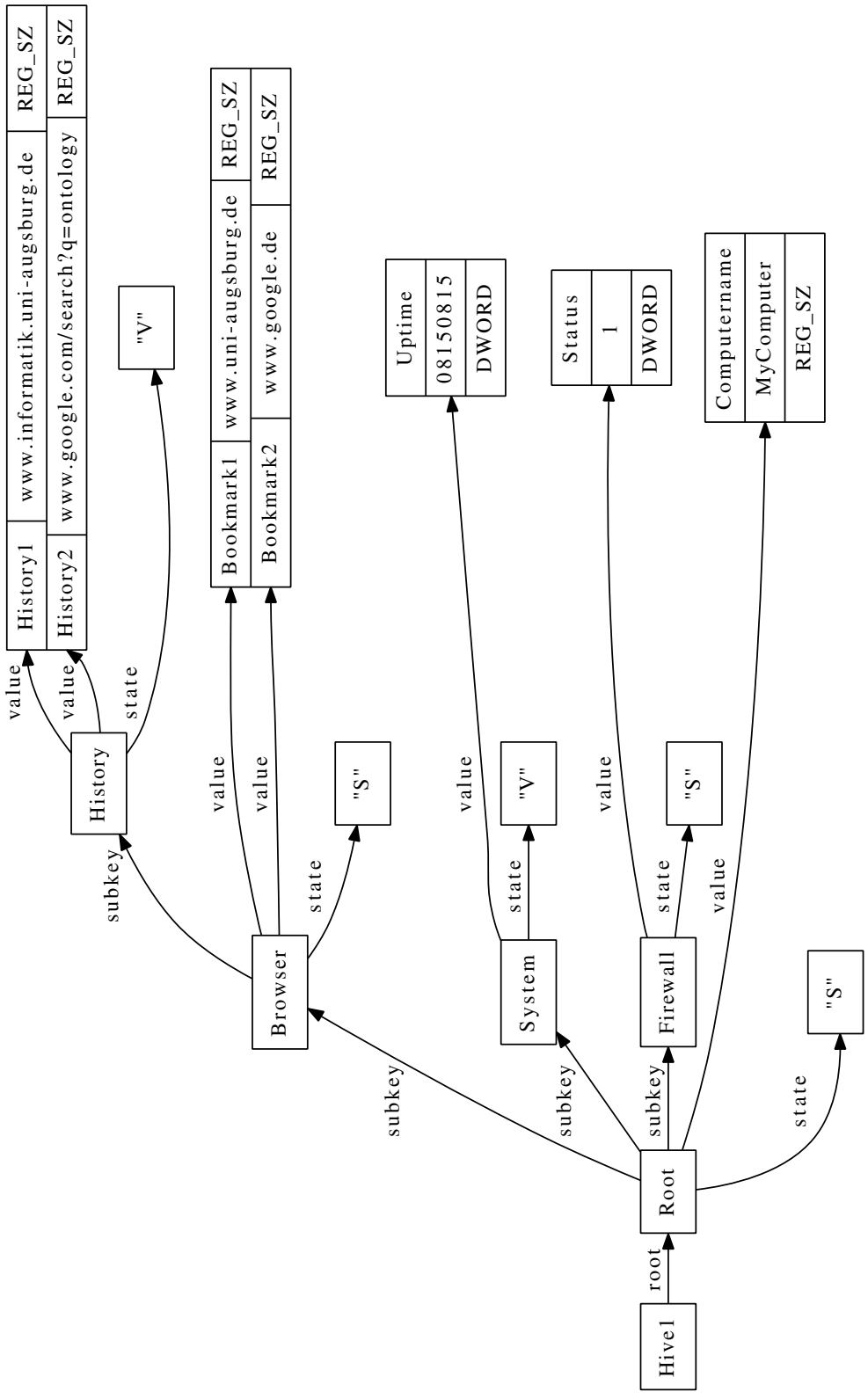

Figure 4.3: Sample registry hive

vulnerable to tampering as any other file. Another problem mentioned in [Kruse and Heiser, 2001, pp 320f] is the admissibility of log files. Depending on the legal boundaries some of them are allowed and others are not. If one needs the information from log files in the ontology one can add it. Another problem of log files is that if they were disabled neither investigators nor tools may be able to detect that or reconstruct the data[Harris, 2006]. Details on adding other data with the example log files can be found in section 7.7.

4.2.4 Example

After the data was collected as explained in section 4.1.3 the next step is to split it up to the structures explained above.

The hard disk data is separated according to section 4.2.1. In the data structure of the file system category all file name entries are listed except the ones for the `ImportantDocument` and the `Malware` file. All file name entries, metadata entries, and all data units are still on the disk, although the files were deleted, and get assigned to the corresponding categories. The data of the `Registry` file is extracted and categorized.

The processes are data in the random access memory that belong to the `Runtime Organization Metadata` category, whereas the handles and connections belong to the according `Data` category and the registry belongs to the `OS-specific Data` category. The registry is extracted and categorized similar to the `Registry` file on the hard disk.

The content of the `Registry` file on the hard disk is shown in figure 4.3. The content if the `Registry` memory object differs from the one on the hard disk only in the `Firewall/Status` which has the value `0` instead of `1`.

How the data fits to the ontology will be explained in section 6.10.

4.3 Forensic Tools

This section provides an overview of the used forensic tools. As for most use cases there are alternatives for these tools. The selected tools are all released under open source licenses. This has some advantage over closed source tools. As explained in [Carrier, 2003] using open source tools, respectively having access to the relevant code for commercial tools, simplifies the procedure of proving the admissibility of the found evidence. In [Manson et al., 2007] open source forensic tools are compared to a proprietary one. It shows that the open source tools are robust and easy to use.

This section is not intended to be a manual for the tools and all their options, thus it will discuss only those parts that are useful for this work.

4.3.1 The Sleuth Kit

The Sleuth Kit[Carrier, 2012a] is a collection of programs for analysing the data on a hard disk respectively on an image of it[Carrier, 2012b]. It was developed by Brian Carrier, author of [Carrier, 2005]. The programs are subdivided into categories. The tools are subdivided according to the categories that are shown in section 4.2.1:[Carrier, 2012c]

- File System

  fsstat Shows file system details and statistics including layout, sizes, and labels. An example output is shown in the appendix in listing B.1.

- File Name

  fls Lists allocated and deleted file names in a directory.

- Metadata

  icat Extracts the data units of a file, which is specified by its meta data address (instead of the file name).

Another set of tools can be used to extract the file system structure from a disk or disk image.

- **mmls** Displays the layout of a disk, including the unallocated spaces.

For most commands an offset of the beginning of the image is required, as the image does not necessarily start with the first partition or the partition one wants to analyse. The `mmls` command displays the structure of the image and the required offset can be read from the output. The output for the example of section 3.2 is similar to the example given in listing 4.1. In this example the first partition starts at 63. The tools for extracting the information from this partition need to be started with `-o 63`.

```
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
00:  Meta    0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000000   0000000062   0000000063   Unallocated
02:  00:00   0000000063   0020948759   0020948697   NTFS (0x07)
03:  -----   0020948760   0020971519   0000022760   Unallocated
```

Listing 4.1: Sample output of mmls

The `fls -m "" -r` command recursively prints all file name entries of the partition in *body file* format. *Body file* format, which can be seen in listing 4.2, is an output format of many of the *Sleuth Kit* tools. Further details about the format are explained in [Carrier, 2009].

Sample output for this command can be seen in listing 4.3.

```
MD5|name|inode|mode_as_string|UID|GID|size|atime|mtime|ctime|crtime
```

Listing 4.2: Body file format

```
MD5|name|inode|mode_as_string|UID|GID|size|atime|mtime|ctime|crtime
0|/$AttrDef|4-128-4|r/rr-xr-xr-x|48|0|2560|1348143371|1348143371|1348143371|1348143371
0|/$BadClus|8-128-2|r/rr-xr-xr-x|0|0|0|1348143371|1348143371|1348143371|1348143371
0|/$BadClus:$Bad|8-128-1|r/rr-xr-xr-x|0|0|10725732352|1348143371|1348143371|1348143371|1348143371
0|/$Bitmap|6-128-1|r/rr-xr-xr-x|0|0|327328|1348143371|1348143371|1348143371|1348143371
0|/$Boot|7-128-1|r/rr-xr-xr-x|48|0|8192|1348143371|1348143371|1348143371|1348143371
0|/$Extend|11-144-4|d/dr-xr-xr-x|0|0|344|1348143371|1348143371|1348143371|1348143371
0|/$LogFile|2-128-1|r/rr-xr-xr-x|0|0|55738368|1348143371|1348143371|1348143371|1348143371
0|/$MFT|0-128-1|r/rr-xr-xr-x|0|0|10829824|1348143371|1348143371|1348143371|1348143371
0|/$MFTMirr|1-128-1|r/rr-xr-xr-x|0|0|4096|1348143371|1348143371|1348143371|1348143371
0|/$Secure:$SDS|9-128-8|r/rr-xr-xr-x|0|0|286944|1348143371|1348143371|1348143371|1348143371
0|/$Secure:$SDH|9-144-11|r/rr-xr-xr-x|0|0|112|1348143371|1348143371|1348143371|1348143371
0|/$Secure:$SII|9-144-14|r/rr-xr-xr-x|0|0|104|1348143371|1348143371|1348143371|1348143371
0|/$UpCase|10-128-1|r/rr-xr-xr-x|0|0|131072|1348143371|1348143371|1348143371|1348143371
0|/$Volume|3-128-3|r/rr-xr-xr-x|48|0|0|1348143371|1348143371|1348143371|1348143371
0|/AUTOEXEC.BAT|7464-128-1|r/rrwxrwxrwx|0|0|0|1348136793|1348136793|1348136793|1348136793
0|/boot.ini|3520-128-3|r/rr-xr-xr-x|0|0|211|1348136591|1348136591|1348136807|1348143500
0|/bootfont.bin|1860-128-3|r/r--x--x--x|0|0|4952|1348143470|1208174400|1208174400|1208174400
0|/CONFIG.SYS|7463-128-1|r/rrwxrwxrwx|0|0|0|1348136793|1348136793|1348136793|1348136793
0|/Dokumente und Einstellungen|3526-144-6|d/drwxrwxrwx|0|0|56|1348137244|1348137244|1348137244|1348140015
0|/IO.SYS|7465-128-1|r/r--x--x--x|0|0|0|1348136793|1348136793|1348136793|1348136793
0|/MSDOS.SYS|7466-128-1|r/r--x--x--x|0|0|0|1348136793|1348136793|1348136793|1348136793
0|/NTDETECT.COM|3492-128-3|r/r--x--x--x|0|0|47564|1348143497|1208174400|1348140049|1208174400
0|/ntldr|3488-128-3|r/r--x--x--x|0|0|251712|1348143497|1208174400|1348140049|1208174400
0|/pagefile.sys|27-128-1|r/rr-xr-xr-x|0|0|402653184|1348137238|1348137238|1348137238|1348143377
0|/Programme|4017-144-6|d/d-wx-wx-wx|0|0|56|1348137144|1348137144|1348137144|1348140039
0|/System Volume Information|3529-144-6|d/dr-xr-xr-x|0|0|56|1348137088|1348137088|1348137088|1348140015
0|/WINDOWS|28-144-6|d/drwxrwxrwx|0|0|56|1348137142|1348137135|1348137135|1348143377
0|/$OrphanFiles|10576|d/d---------|0|0|0|0|0|0
```

Listing 4.3: Sample output of fls

4.3.2 Volatility

Volatility[Volatile Systems, 2012b] is a framework for analysing random access memory dumps. For this work version 2.1 is being used. One can specify the path to the image to use in the command line or export it to the environment. The program is started with `python vol.py` and appended parameter defining which modules to run. The modules used in the implementation can be grouped as follows [Volatile Systems, 2012a]:

- Processes

  pslist Lists processes that are in the doubly-linked list of the operating system.

psscan Use pool tag scanning to find processes that are not necessarily in the list of the operating system. Details to this technique can be found in [Schuster, 2006] and [Van Baar et al., 2008].

psdispscan Similar to psscan but with different memory structure to look for.

thrdscan Similar to psscan but searching for threads.

envars Lists the environment variables associated with a process.

getsids Lists security identifiers associated with a process. Useful for detecting privilege escalation.

handles Lists the handles associated with a process.

dlllist Lists the dynamic link libraries associated with a process.

- Networking

connections Lists the TCP connections that can be found in the singly-linked list of the operating system.

connscan Scans for TCP connections or fragments of connection date in the memory.

sockets Similar to connections but for all protocols.

sockscan Similar to connscan but for all protocols.

- Registry

hivelist Lists the available hives and their location in the memory and on hard disk. An example output is shown in the appendix in listing B.2.

hivedump Lists all subkeys in a specified hive.

printkey Prints the information stored at a specific key. If no hive is provided and the key exists in more than one hive, the information of all hives is printed.

hivedump2 Custom module that combines hivedump and printkey functionality. Details are described in section 7.4.

The modules that contain *"scan"* in their name search the memory for data patterns that indicate the relevant data structures. If there are multiple modules that search for the same objects, for example *pslist*, *psscan* and *psdispscan*, the differences between the results can indicate that something might have been manipulated, for example that malware tries to hide from the operating systems process list.

As the output of the tools of the different categories looks similar only one output is shown as example. A sample output of *psscan* from the processes section is shown in listing 4.4. For the networking category the output

of the *sockets* command is displayed in listing 4.5. Last but not least listing 4.6 demonstrates the output of *printkey* from the registry category.

```
Volatile Systems Volatility Framework 2.1
Offset(P)  Name                PID   PPID PDB         Time created          Time exited
---------- ---------------- ------ ------ ---------- -------------------- --------------------
0x018312a0 ctfmon.exe         1168    408 0x0ee7c000 2012-09-20 10:34:18
0x01898a20 explorer.exe        408    364 0x0b23e000 2012-09-20 10:34:13
0x0189eda0 wscntfy.exe         316    976 0x0b074000 2012-09-20 10:34:13
0x018b3880 alg.exe            2032    636 0x0ab8e000 2012-09-20 10:34:12
0x01934148 spoolsv.exe        1364    636 0x08b8f000 2012-09-20 10:34:00
0x0193e2c8 wpabaln.exe        1044    592 0x0d8f1000 2012-09-20 10:36:13
0x01962c78 svchost.exe        1088    636 0x06a52000 2012-09-20 10:33:59
0x0196a8b0 svchost.exe        1036    636 0x067e8000 2012-09-20 10:33:59
0x01972408 svchost.exe         976    636 0x0658a000 2012-09-20 10:33:59
0x0197fbd0 svchost.exe         884    636 0x06334000 2012-09-20 10:33:59
0x019a1a70 svchost.exe         804    636 0x05d50000 2012-09-20 10:33:59
0x019bc3f0 lsass.exe           648    592 0x052dc000 2012-09-20 10:33:58
0x019bfc50 services.exe        636    592 0x0526e000 2012-09-20 10:33:58
0x019d5788 csrss.exe           568    504 0x04520000 2012-09-20 10:33:58
0x019e87c0 winlogon.exe        592    504 0x048a6000 2012-09-20 10:33:58
0x01a2c990 smss.exe            504      4 0x03404000 2012-09-20 10:33:58
0x01bcca00 System                4      0 0x00039000
```

Listing 4.4: Sample output of psscan

```
Volatile Systems Volatility Framework 2.1
Offset(V)      PID    Port  Proto Protocol        Address          Create Time
---------- ------ ------ ------ --------------- --------------- -----------
0x814f76b8    648    500     17 UDP             0.0.0.0         2012-09-20 10:34:09
0x816353b8      4    445      6 TCP             0.0.0.0         2012-09-20 10:33:58
0x8157a560    884    135      6 TCP             0.0.0.0         2012-09-20 10:33:59
0x814abb00   2032   1025      6 TCP             127.0.0.1       2012-09-20 10:34:13
0x814c6708    976    123     17 UDP             127.0.0.1       2012-09-20 10:34:28
0x814f5e98    648      0    255 Reserved        0.0.0.0         2012-09-20 10:34:09
0x8152a008   1088   1900     17 UDP             127.0.0.1       2012-09-20 10:34:28
0x814f6710    648   4500     17 UDP             0.0.0.0         2012-09-20 10:34:09
0x816355f0      4    445     17 UDP             0.0.0.0         2012-09-20 10:33:58
```

Listing 4.5: Sample output of sockets

```
Volatile Systems Volatility Framework 2.1
Legend: (S) = Stable   (V) = Volatile

----------------------------
Registry: \Device\HarddiskVolume1\Dokumente und Einstellungen\LocalService\NTUSER.DAT
Key name: Run (S)
Last updated: 2012-09-20 10:31:15

Subkeys:

Values:
REG_SZ        CTFMON.EXE      : (S) C:\WINDOWS\system32\CTFMON.EXE
```

Listing 4.6: Sample output of printkey

4.3.3 reglookup

The further source for registry information are the hive files on hard disk. *reglookup*[Sentinel Chicken Networks, 2010] is used to extract the registry information from the registry files on the hard disk. The cropped output of *reglookup* that corresponds to the one of *printkey* is shown in listing 4.7.

```
PATH,TYPE,VALUE,MTIME,OWNER,GROUP,SACL,DACL,CLASS
/Software/Microsoft/Windows/CurrentVersion/Run/CTFMON.EXE,SZ,C:\WINDOWS\system32\CTFMON.EXE,,,,,
```

Listing 4.7: Sample output of reglookup

4.3.4 bkhive + samdump2

The two tools *bkhive*[Tissieres and Oechslin, 2013] and *samdump2*[Tissieres and Oechslin, 2013] are used to extract information about the user. The output of *bkhive* is given to *samdump2* and the result is shown in listing 4.8.

```
Administrator:500:6a98eb0fb88a449cbe6fabfd825bca61:a4141712f19e9dd5adf16919bb38a95c:::
Gast:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
Hilfeassistent:1000:50a75aa3555c00d0ba0322f551cc115a:afacea076c4a025a3022c614793f9e46:::
SUPPORT_388945a0:1002:aad3b435b51404eeaad3b435b51404ee:a484598dba956d06f2a8fc23c14d2c92:::
Benutzer1:1003:d7246e4feea4219d179b4d5d6690bdf3:9068eeaf33cffd1d86ac515e518588a0:::
```

Listing 4.8: Sample output of samdump2

Chapter 5

Ontology

This chapter will explain the technical aspects of ontologies. Thus in this chapter "ontology" means the common ontology and not the introduced one.

5.1 Ontology Basics

An ontology is the representation of knowledge of a specific domain and the connections between the concepts found in the knowledge. Furthermore an ontology allows reasoning about the stored information.[Gruber, 2009]

In [Noy and McGuinness, 2001] basics for developing ontologies are explained. Regarding the reasons that speak for using an ontology, the primary one that matches the goal of this work is that it can be used to "analyse domain knowledge"[Noy and McGuinness, 2001] as we want to draw conclusions about the degree of infection of a computer which is an instance of the ontology. Another aspect that applies is that of the re-usability because the structure of computer systems do not differ very much. An ontology consists of classes, properties, and restrictions. These terms stand for concepts that may be called other names at different authors and technologies. Instances are specific objects of the concept described by the class. For the class "diploma thesis" this work is an instance. When instances of classes are added this is called a knowledge base. However the distinction between ontology and knowledge base is difficult as some instances may be needed to describe the concept.

Although many concepts originate from object oriented design, the big difference is that ontologies are designed "based on the structural properties of a class"[Noy and McGuinness, 2001] whereas the object oriented designs are "based on the operational properties of a class "[Noy and McGuinness, 2001]. Classes stand for concepts and are preferably designed close to the objects of the domain. As an example an address book is being modelled. Some possible concepts that do not necessarily become classes in the end are person, name, title, address and telephone number. Properties describe

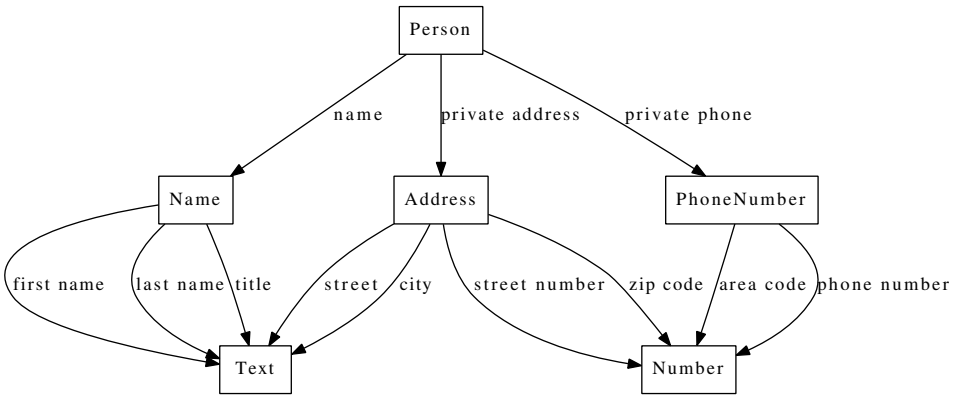

Figure 5.1: Address book ontology

the characteristics of the concepts. In the address book example the first and last name are properties of the class name and street, city and zip code are properties of the address. The restrictions constrain the applicability of the properties. The properties can be thought of as the edges that connect the nodes of a directed graph. The restrictions limit what types of edges are allowed between what types of nodes. In the example it would not make any sense if the edge that represents the property for the first name would be allowed to connect nodes of the type address and person. One possibility for the address book ontology is shown in figure 5.1. An instance of the structure of the example ontology that describes the address book entry of a specific person may look similar to figure 5.2.

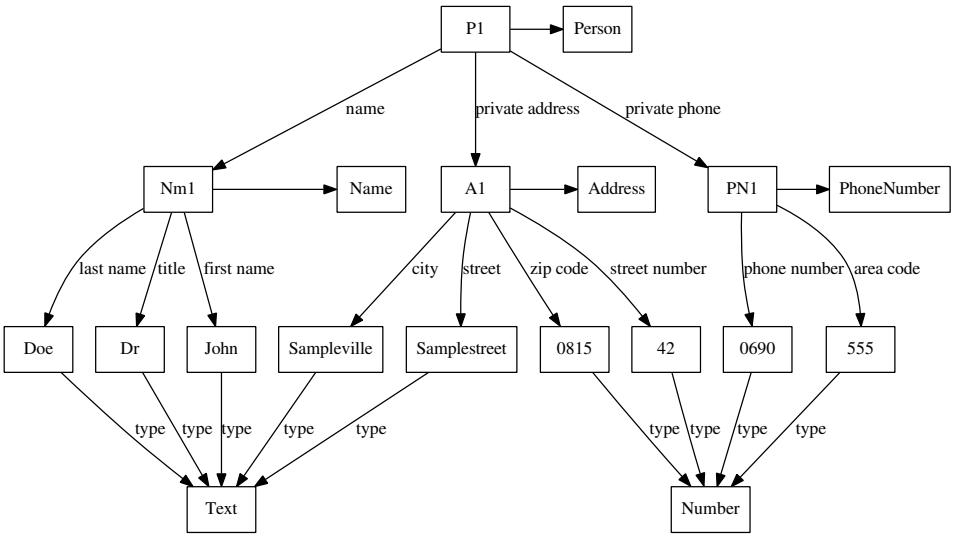

Figure 5.2: Address book instance

To store an ontology as text it can be described by triples. Triples consist of subject, predicate and object. For example in $User1|name|Administrator$ $User1$ is the subject *name* is predicate and *Administrator* is object. A text representation of the ontology from figure 5.1 is shown in listing 5.1. The instance from figure 5.2 can be written with tuples as shown in listing 5.2.

```
Subject          Predicate              Object
"Person"         "name"                 "Name"
"Name"           "first name"           "Text"
"Name"           "last name"            "Text"
"Name"           "title"                "Text"
"Person"         "private address"      "Address"
"Address"        "street"               "Text"
"Address"        "street number"        "Number"
"Address"        "city"                 "Text"
"Address"        "zip code"             "Number"
"Person"         "private phone"        "PhoneNumber"
"PhoneNumber"    "area code"            "Number"
"PhoneNumber"    "phone number"         "Number"
```

Listing 5.1: Address book triples

```
Subject          Predicate              Object
"P1"             "type"                 "Person"
"P1"             "name"                 "Nm1"
"Nm1"            "type"                 "Name"
"Nm1"            "first name"           "John"
"John"           "type"                 "Text"
"Nm1"            "last name"            "Doe"
"Doe"            "type"                 "Text"
"Nm1"            "title"                "Dr"
"Dr"             "type"                 "Text"
"P1"             "private address"      "A1"
"A1"             "type"                 "Address"
"A1"             "street"               "Samplestreet"
"Samplestreet"   "type"                 "Text"
"A1"             "street number"        "42"
"42"             "type"                 "Number"
"A1"             "city"                 "Sampleville"
"Sampleville"    "type"                 "Text"
"A1"             "zip code"             "0815"
"0815"           "type"                 "Number"
"P1"             "private phone"        "PN1"
"PN1"            "type"                 "PhoneNumber"
"PN1"            "area code"            "555"
"555"            "type"                 "Number"
"PN1"            "phone number"         "0690"
"0690"           "type"                 "Number"
```

Listing 5.2: Address book instance triples

5.1.1 Creating an ontology

This section describes one possible approach to develop an ontology. It is mainly based on [Noy and McGuinness, 2001] and the author's experiences when creating the forensic ontology.

There is not the one perfect way to do this. The method depends on the application it is designed for and the possible extensions that should be possible. It is recommendable to use the technical terms of the domain in the ontology. A guideline which steps should be taken is outlined in the following.

1. Identify the domain and the scope. This includes finding out which domains the ontology should deal with. Another aspect is the field it will be used for and the questions that should be answered. An important point is the person to use and maintain the ontology. For the forensic ontology these questions are answered in chapters 3 and 4.

2. Can existing ontologies or structuring be reused? This issue is important in many ways. If an ontology exists, it saves much time as it can be used as it is or can be adapted. If there is a structuring for data that should be used, this can be a lead for the creation of the ontology. In the current case for the hard disk and the random access memory structures already exist, as explained in section 4.2.1 and 4.2.2.

3. List all terms that belong to the subject. It is useful to write down all terms regardless of implied structure so that nothing is forgotten.

4. Define classes and class hierarchy, properties of the classes and restrictions of the properties. This step first requires to realize which terms should be represented as classes and which as properties. This question will be explained more thoroughly at the end of this section. As the next step, the classes, then the properties, and finally the restrictions need to be identified. The restrictions can be compared to data types in typed programming languages. It has to be decided about the representation the data types need to have in the ontology. For example, the property that identifies the forensic tool a piece of information was retrieved by needs a restriction to such an extent that the value needs to be of the type "forensic tool".

5. Build instances of the specified classes. As a last step, instances belonging to the ontological side of the structure need to be specified. Whether an instance is part of the ontology or part of the data that is stored is in the eye of the beholder. For example, the specific forensic tools can be counted to the ontology because they are known prior to the examination. On the other hand it can be argued that the ontology has to be independent of the tools and the tools belong to the data that is identified in the course of the examination.

After all this is done, the ontology can be revised for consistency. At first, it can be ensured that the hierarchy is correct. Next it can be checked that the subclasses are transitive. Furthermore it can be helpful if subclasses are disjoint.

If the new subclass has properties that the superclass does not have, the term should be represented as a new class rather than as a property. A similar question is whether a term should be a class or an instance. For example, the tools might be separate classes with the same argumentation as above.

5.1.2 Advantages of ontologies for forensics

The ontological approach is used because the connections between the different pieces of data described in section 4.2 create a directed graph structure. The connections link different types of data. For example, a process that resides in random access memory is created from running a program that is located on the hard disk. This process can have handles to other files and to network connections. Thus there is a connection between the random access memory and the hard disk for each file referenced by a process, and connections between random access memory and the network interface for the network connections.

Another point for using the ontology based approach is that the corresponding query language allows queries to stay the same, independent of the data as long as the structure remains the same. As the resulting data is a graph, cross references between different bits of data can be represented directly as edge between the corresponding nodes. These connections can also easily be added later.

5.1.3 XML/RDF(S)/OWL

This section explains the different technologies that can be used to represent an ontology.

Extensible Markup Language (XML) is a hierarchical text representation for data. It was invented for easy data exchange between different computer systems. XML documents have a tree structure. [Hitzler et al., 2008a]

Resource Description Framework (RDF) defines a format for describing logical expressions over resources. It is a basic technology for the semantic web. RDF is an extension of XML, so RDF documents also have a tree structure. But RDF documents can describe data that has a directed graph structure.[Hitzler et al., 2008b]

Resource Description Framework Schema (RDFS) is a format for defining the structure of RDF documents. In our case this is used to describe the ontology.[Hitzler et al., 2008b]

Web Ontology Language (OWL) is a format for creating ontologies. It is based on first order logics. OWL has the three sub languages Lite, DL and Full. The information about OWL is taken from [Hitzler et al., 2008c] and [W3C, 2004]. OWL can be serialized with XML[W3C, 2009b] or RDF[W3C, 2009a] syntax.

OWL Lite was designed to make it easy to implement a subset of the language. It forbids for example the use of owl:complementOf.

OWL DL OWL-DL requires classes, properties individuals and data values to be disjoint[W3C2004]. In contrast to OWL Full, it is decidable.

OWL Full allows constructions with RDFS elements and loosens the restrictions from OWL DL which causes that OWL Full is undecidable.

Summary: OWL Lite was designed to provide an easy beginning in learning and implementing OWL. From OWL Full over OWL DL to OWL Lite more and more restrictions are specified. The restrictions include specification which predicates are allowed for usage, which types to apply to a predicate and what is allowed to be placed at the different places of triples. For example, in OWL Full owl:Class and rdfs:Class are equivalent. In OWL DL and OWL Lite owl:Class is a subclass of rdfs:Class[W3C, 2004].

In our case the RDF files contain the data obtained from the computer. Chapter 6 describes how all the information is stored in this format.

The target of RDFS and OWL is to allow a description of relations between parts of information in a manner that a computer can derive new information from that. For this work, RDF was chosen instead of OWL. One reason for that decision is that OWL full, which allows to model as much as RDFS and more, is undecidable. This has the consequence that one cannot say whether an issued query can be answered according to [Hitzler et al., 2008c]. In contrast, OWL DL is designed that the question whether a statement can be deduced from the ontology is ensured to be answerable. Another reason is that RDF(S) is supported by more tools and thus is easier to use.

5.1.4 Storage

For storing the ontology data some sort of database is used preferably. Graph databases can be used because the structure of the ontological data is a

graph. Alternatively a triple store can be used because the data can be written as triples. Both of these possibilities can be implemented on a relational database. Some graph database frameworks already support the use of relational databases as storage. Two databases that can be used for storing ontology data are presented in section 5.3.

5.1.5 SPARQL

SPARQL is a query language. Queries are questions that are asked to a database. The syntax of SPARQL is similar to SQL. With SPARQL the ontology can be queried for the required information. In contrast to SQL, SPARQL queries for example do not need to take precautions for possible empty results[Prud'hommeaux, 2012].

SPARQL is used to query the RDF data. A simple query is provided in listing 5.3. The query asks for the last names of all people who have the first name Jan. Line one defines the abbreviations of the namespaces. Namespaces specify the locations where data structures are defined. For example in `http://www.example.org/person` the structure of the data type person is defined. Line two of the query specifies that the variable `?lastname` is the output. The lines three to six specify the constraints of the query. In line four it is specified that there is a variable `?person` and an object, that is the value of `?person`, has to contain a variable called `p:firstname`, which has the value `Jan`. The `p:firstname` tells that the property `firstname` is defined in the namespace that is abbreviated to `p`. Line five defines that the variable `?lastname` contains the value of the variable `p:lastname` of the same object, that is called `?person`, as in line four. The lines of the `WHERE` block resemble the triple structure as explained in the beginning of section 5.1.

```
1 PREFIX p:<http://www.example.org/person#>
2 SELECT ?lastname
3 WHERE {
4 ?person p:firstname "Jan" .
5 ?person p:lastname ?lastname .
6 }
```

Listing 5.3: Simple SPARQL query

A query a bit more advanced is provided in listing 5.4. It asks for the names of all people who live in Augsburg. The lines one and three to six are similar to the query in listing 5.3 except that the first name is also a variable that is contained in the output. In line seven an additional variable called `?address` is defined that contains the value of `p:address` of `?person`. Line eight says that the value of the variable `a:city` of the variable `?address` has

to be `Augsburg`.

The query from listing 5.4 can be written shorter. The variable `?address` is only used to store the address object temporarily and is not needed anywhere else. In most programming languages the code `address = person.address; address.city == "Augsburg"` can be replaced by `person.address.city == "Augsburg"`. This is similarly also possible in SPARQL as it is shown in listing 5.5.[Pérez et al., 2009][W3C, 2008]

More details on writing queries are given in section 7.6.

```
1 PREFIX p:<http://www.example.org/person#>
2 PREFIX a:<http://www.example.org/address#>
3 SELECT ?lastname ? firstname
4 WHERE {
5 ?person p:firstname ?firstname .
6 ?person p:lastname ?lastname .
7 ?person p:address ?address.
8 ?address a:city "Augsburg" .
9 }
```

Listing 5.4: Advanced SPARQL query

```
1 PREFIX p:<http://www.example.org/person#>
2 PREFIX a:<http://www.example.org/address#>
3 SELECT ?lastname ? firstname
4 WHERE {
5 ?person p:firstname ?firstname .
6 ?person p:lastname ?lastname .
7 ?person p:address [ a:city "Augsburg" ] .
8 }
```

Listing 5.5: Advanced SPARQL query (shortened)

5.2 Ontology Tools

In this section tools are presented for creating and visualizing ontologies. As last section a conclusion of the tools is given.

5.2.1 Altova Semantic Works

Altova Semantic Works[Altova, 2013] is a tool to create RDF(S) and OWL files. Class hierarchies can be created graphically. It allows to use files as a

resource for the namespaces what makes it easier to split the ontology into multiple files.

5.2.2 Protégé

"*Protégé* is a free, open source ontology editor and knowledge-base framework."[Stanford Center for Biomedical Informatics Research, 2013] This ontology editor is mainly built for using OWL files. It can be used for creating RDFS files since OWL supports RDFS elements and is technically also an XML extension.

5.2.3 Gephi

"*Gephi* is an interactive visualization and exploration platform for all kinds of networks and complex systems, dynamic and hierarchical graphs."[Gephi Consortium, 2012] The *SemanticWeb* plugin allows to import RDF(S) files via SPARQL `CONSTRUCT` queries. It implements several graph layout algorithms and renders nice graphics of the input. A screenshot of the interface can be found in figure C.3.

5.2.4 RDF Gravity

"*RDF Gravity* is a tool for visualising RDF/OWL Graphs/ ontologies." [Salzburg Research, 2012] The implemented filters allow a very fast graphical overview of the RDFS files. A screenshot of the interface can be found in figure C.2.

5.2.5 Cytoscape

Cytoscape[Cytoscape Consortium, 2012] is another tool for visualizing network data. A plugin is needed to import RDF data. Similar to *Gephi* a SPARQL `CONSTRUCT` or `DESCRIBE` query is needed for importing.

5.2.6 Conclusion

For creating the RDFS files *SemanticWorks* was used because it is easier to use and has a much clearer interface than *Protégé*.

If the ontology is small or split up into several small files, as it is in this work, the simplest way to visualize the ontology is by *RDF Gravity*. For *Gephi* and *Cytoscape* plugins are available for importing RDF(S) files. At first the source of the data has to be specified and then it can be imported via a SPARQL query. In all three tools, the nodes have to be distributed after loading the data to get an overview. *RDF Gravity* has the least advanced layout algorithm but responds the fastest. Within the other two tools it is much more complicated to get a decent result.

Gephi and *Cytoscape* visualization tools might provide a nicer output but it takes much more time to get them to do what is wanted.

5.2.7 Raptor RDF and GraphViz

There is another possibility to create graphical representations of the ontology. *Raptor RDF*[Beckett, 2013] is a RDF parser that can output the data of the RDF file in the dot format of *GraphViz*[Ellson et al., 2013]. *GraphViz* then converts the dot file to an image file. For example, the command `rapper -I . -o dot sample.rdf | dot -Tpng -o sample.png` converts a RDF file named `sample.rdf` to a PNG image file called `sample.png`. All figures in this work that show graphs or graph-like structures are generated from dot files.

5.3 Storage

This section presents the two storage possibilities that were used when creating the example implementation. An explanation why there are two and which ones were used in the end is given in section 7.5.

5.3.1 Neo4J

Neo4J[Neo Technology, Inc, 2013] is a graph database that is implemented in Java. Graph means property graph. It consists of nodes and relationships. Both of them have properties and the relationships structure the nodes. This structure is visualized in figure 5.3. According to [Neo Technology, Inc., 2006] relational databases do not support the recently upcoming amount of data that is structured in networks. *Neo4J* is designed to fit the requirements of this kind of data. Additionally, the *Neo4J* database is preferably to be used with semi-structured data. Semi-structured data can be thought of as a table where the entries have few mandatory attributes but many optional ones. But a drawback is that arbitrary queries on structured data are not handled as efficiently as in relational databases. This is caused by the network focused design. *Neo4J* has a graphical web front end which allows interactive browsing of the database. A screenshot of this interface can be found in the appendix in figure C.1.

5.3.2 Sesame

"*Sesame* is an open source Java framework for storage and querying of RDF data."[Aduna, 2012] *Sesame* is a triplestore that is designed for storing and retrieving triples. The web interface allows browsing the stored data and direct SPARQL queries.

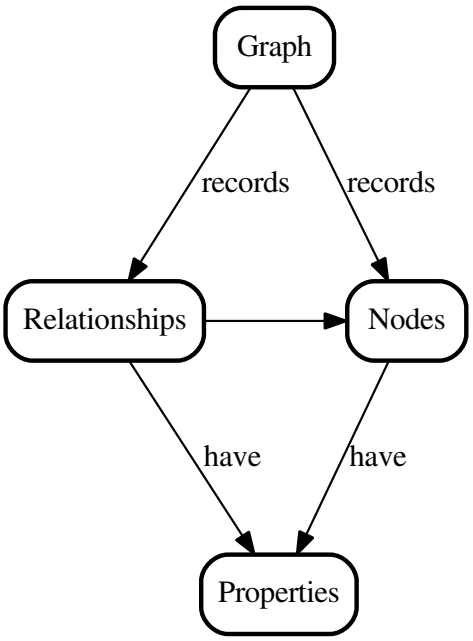

Figure 5.3: Neo4J property graph[Neo Technology, Inc, 2013]

Sesame is an abstracted architecture that allows the usage of different back ends for the storage of the data. Its development helped to uncover and remove unclarities in the RDFS specification.[Broekstra et al., 2002] Details about *Sesame* are provided in section 7.5.

Chapter 6

Forensic Ontology

The ontology represents all forensically relevant parts of a computer at the time the snapshots were taken, limited by the prior mentioned problems, and is stored in RDFS files. The data is structured according to the models in section 4.2. In this chapter the data structures are explained. The description of the structure starts with the small and simple sub-sections of the ontology and later shows the complex parts. Details of the exact implementation are explained in chapter 7.

For easier reading the nodes for rdf:Class and rdf:Property and the corresponding edges for the rdf:type were left out in the figures.

Class names start with capital letters and property names with a small letter. Properties starting with "has" normally have a cardinality greater than one, others normally one or zero. rdf:range defines the data type to which the edge of the property points from and rdf:domain the data type where it points to.

6.1 Forensic Object

First of all, as the data is used for forensics each *ForensicObject* must be associated with the *Timestamp* and the *ForensicTool* it was retrieved by, as it can be seen in figure 6.1. By adding these connections, one can search for information retrieved by one special tool or if there is different information by different tools.

6.2 Hardware

The first things one thinks of when structuring a computer is the hardware. So the hardware part represents the forensically interesting parts of the computer's hardware. As mentioned before the relevant parts are the *Memory*, the *Harddisk*, and the *NetworkInterfaceCard(NIC)* as visible in figure 6.2

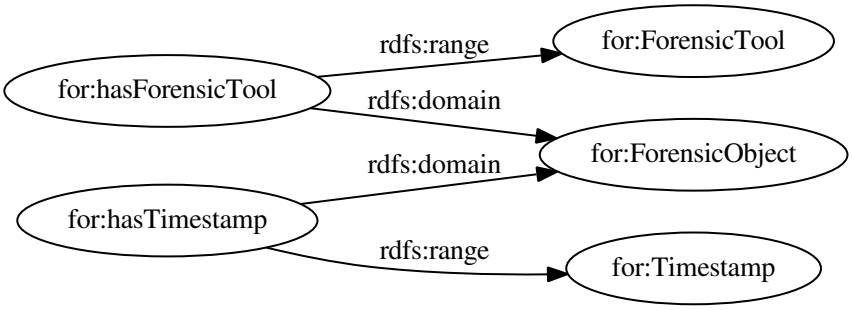

Figure 6.1: Forensic Object

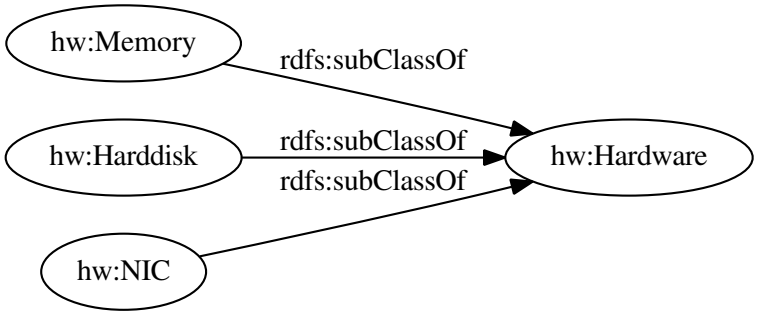

Figure 6.2: Hardware

6.3 Software

The next category that comes to mind when structuring a computer is the software. The generic software parts are the *Kernel*, the *Resources*, and the *ProcessList*. This can be seen in figure 6.3. *Resources* stand for file handles, network connections, and other handles the kernel provides. Details for the *ProcessList* are specified in section 6.5.

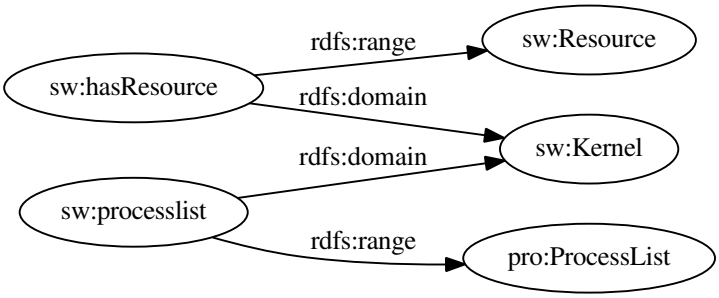

Figure 6.3: Software

6.4 User

A further obvious element is the *User* respectively more of them. The essential data for each of them is the *Name*, a *Password*, and one or more *Groups* the *User* belongs to, as it is visible in figure 6.4.

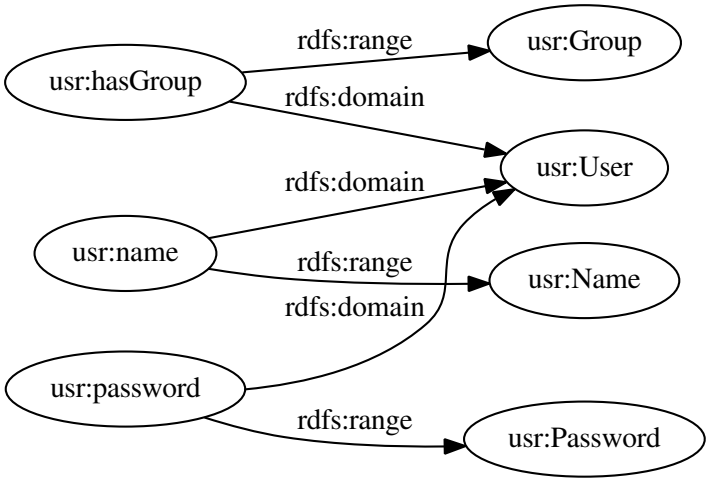

Figure 6.4: User

6.5 Process

Every program that runs on a system has at least one *Process* and all of them are listed in the *ProcessList*. Each of them, except the initial one, has at least one *Thread* and one parent *Process*. The *Resources* from the *Kernel*, defined in section 6.3, can be used by *Processes*. The structure can be seen in figure 6.5. It is assumed that the most basic parent process has itself as a parent.

6.6 Network

One element in the list of the *Hardware* is the *NIC*. It has a *Configuration* which includes *IP*, *Gateway*, and *Nameserver*. On the other hand there are *Connections* associated with it. *Connections* have a *local* and a *remote IP* address. For usage the connections have to be wrapped by *Sockets*. These have a *Port* and a *Protocol* and can be referenced as *Resources* by *Processes*. This structure can be seen in figure 6.6.

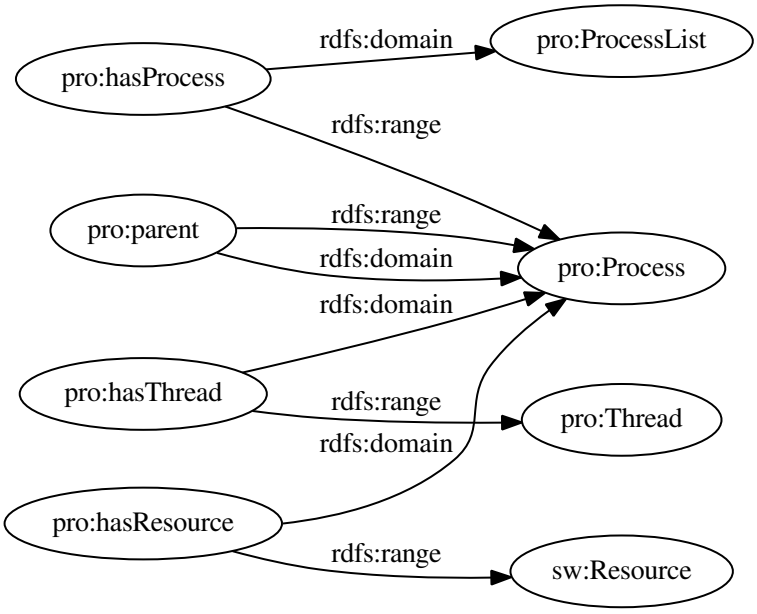

Figure 6.5: Process

6.7 Registry

The representation of the *Registry* in the ontology, as shown in figure 6.7, is very close to the structure described in section 4.2.3.1. The *Registry* contains *Hives* and each of them has a *root Key*. Each *Key* has a name and a *State* that represents the flag where the key can be found. *Keys* can have *sub-Keys* and *Values*. *Values* contain a *key value* pair, a *ValueType* for the type of the stored value and also a *State*.

6.8 File System

A *Harddisk* has a *Partition*. The *Partition* is divided into the five classifications from section 4.2.1. The different categories have their corresponding RDFS class for the single entries. The entries are connected to the *Partition* with the appropriate *has** property. Then there is the meta structure *FileSystemObject* which connects the associated entries of the different sections, but the derived classes *File* and *Folder* preferably should be used as they allow to build the typical file hierarchy. The structure is visualized in figure 6.8.

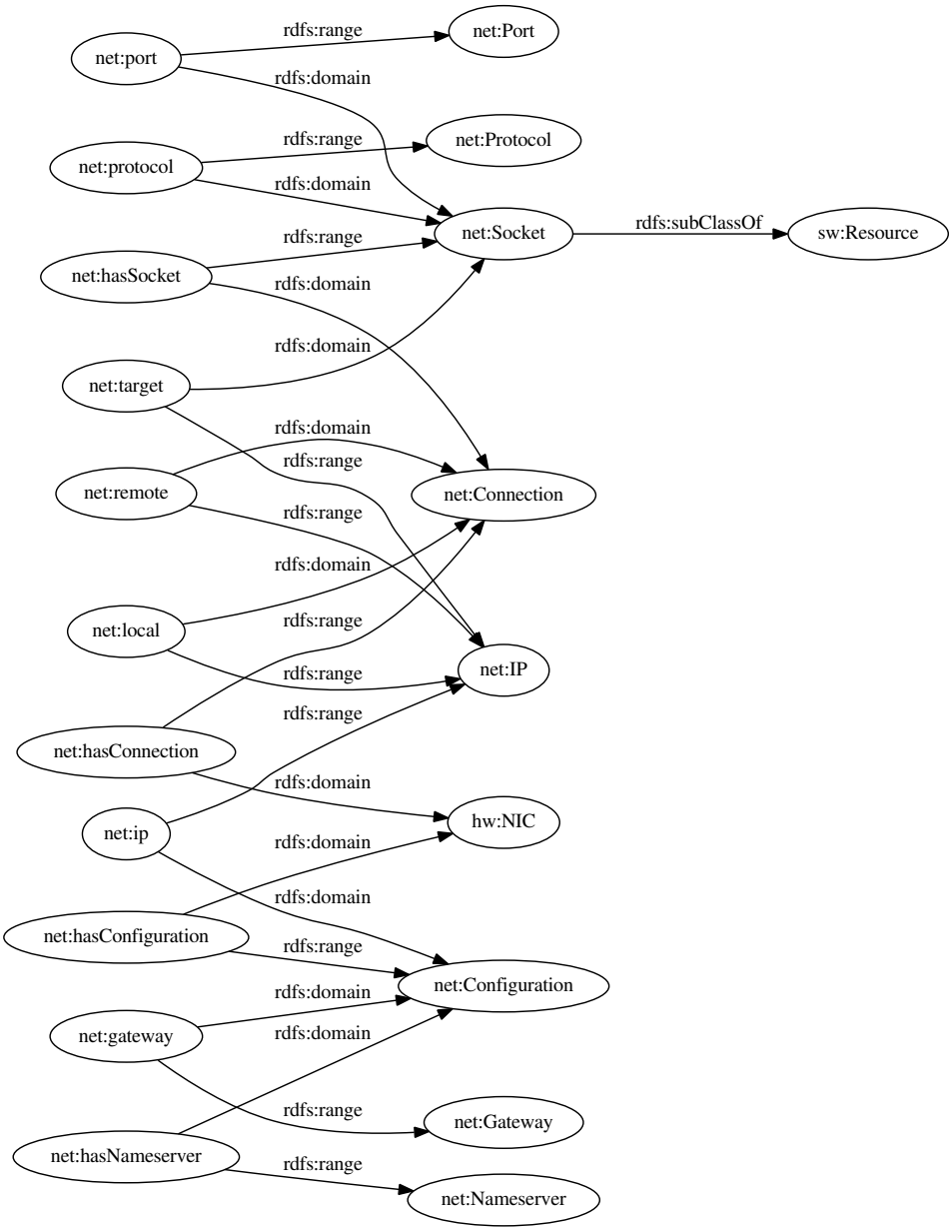

Figure 6.6: Network

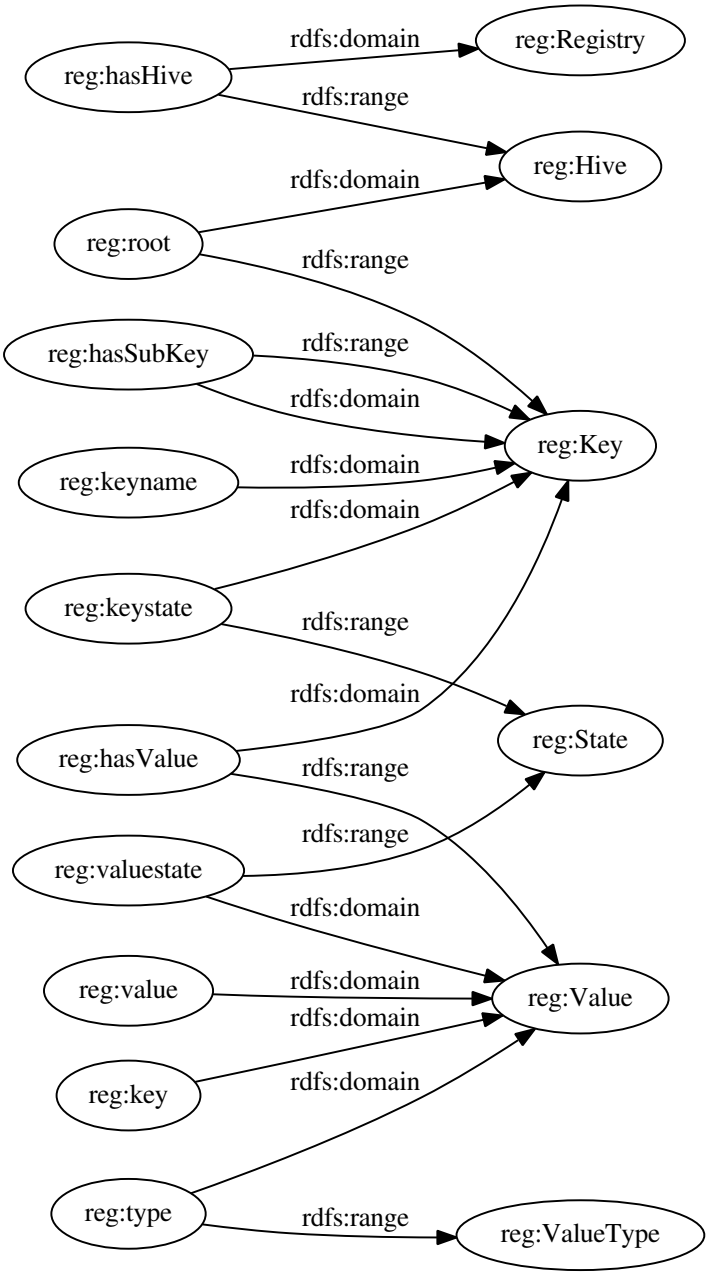

Figure 6.7: Registry

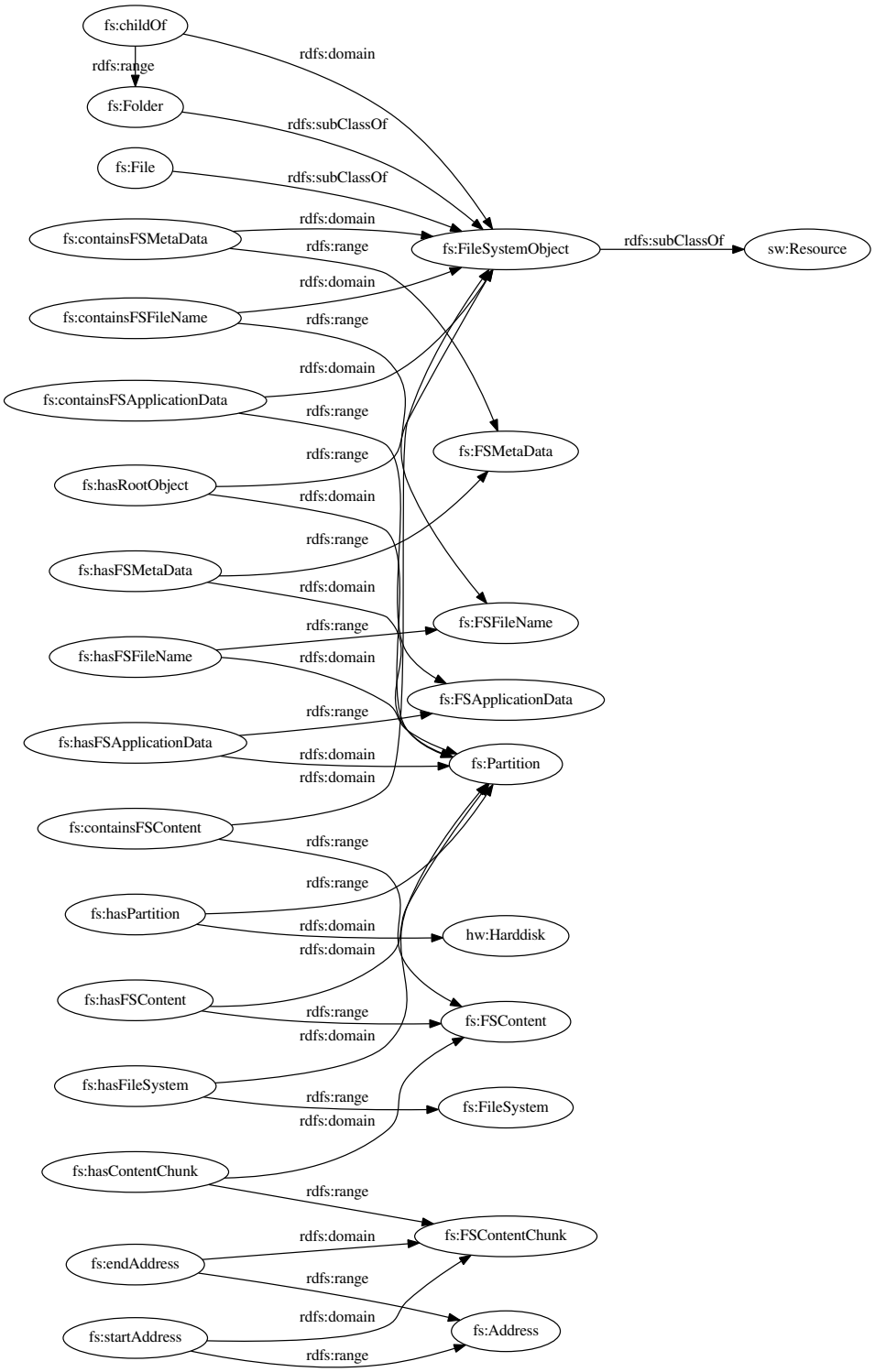

Figure 6.8: File System

6.9 Memory

The *Memory* is structured closely to the description in section 4.2.2 as shown in figure 6.9. The *Memory* associates the entries for *MemorySystemArchitecture* and *Metacode*. The categories *Metadata*, *Data* and *Code* are linked to the corresponding RDFS classes for the specific entry types.

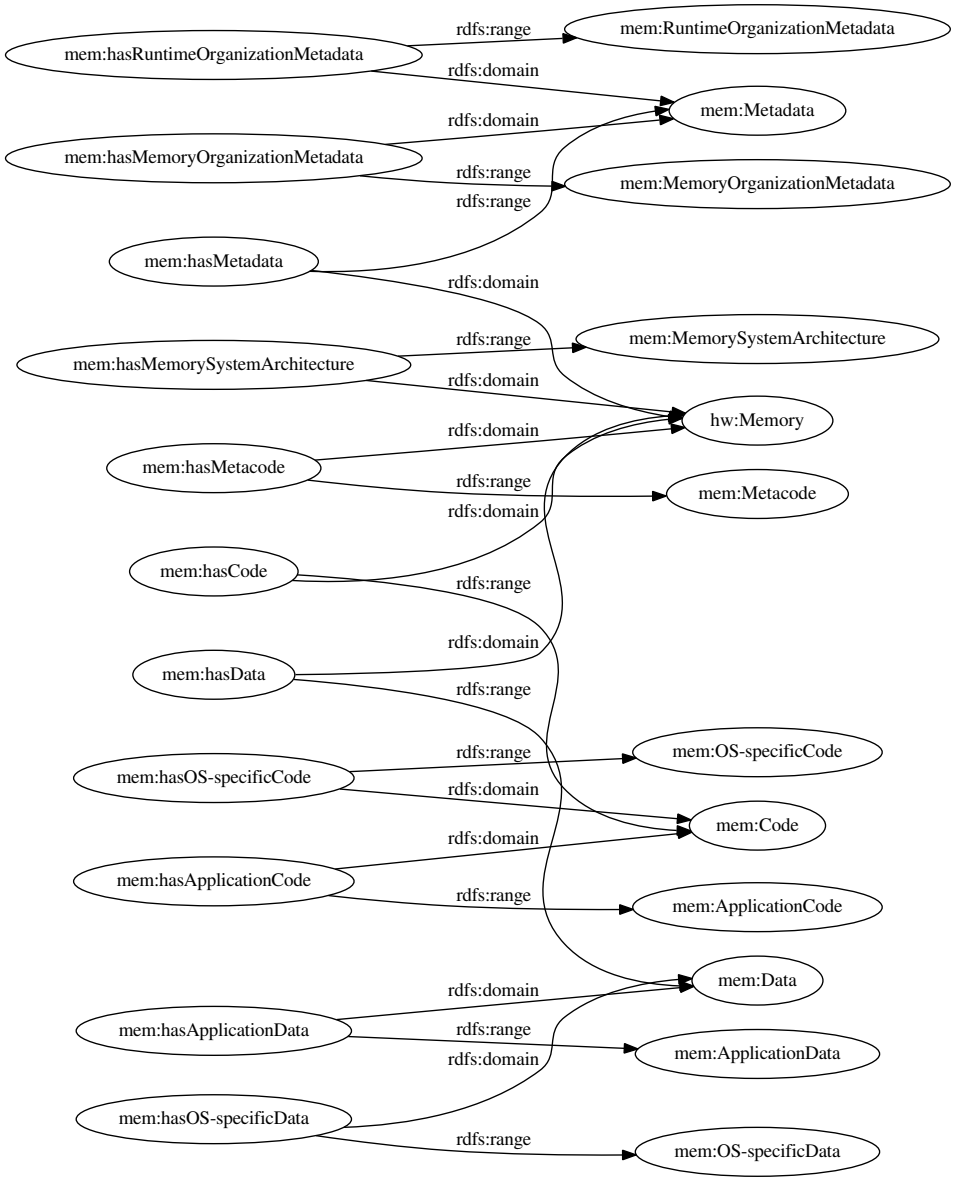

Figure 6.9: Memory

6.10 Example

In section 4.2.4 the data was structured. Now it will be put into the ontology.

6.10.1 Hard Disk

Listing 6.1 shows the triples that represent the data from the hard disk. The structure that is created by the `FileSystemObject` type is similar to the structure that can be seen when normally accessing the hard disk, but the deleted files are listed similarly to the not deleted files because the corresponding `FileName` entry provided the information where the file was located originally.

```
Harddisk1       rdf:type                hw:Harddisk
Harddisk1       fs:hasPartition         Partition1
Partition1      rdf:type                fs:Partition
Partition1      fs:hasRootObject        FsFn0

Partition1      fs:hasFSFileName        FsFn0
FsFn0           rdf:type                fs:FSFileName
FsFn0           fsfn:name               "Root Object"
Fs00            rdf:type                fs:Folder
Fs00            fs:containsFSFileName   FsFn0
Fs00            fs:childOf              Fs00

Partition1      fs:hasFSFileName        FsFn1
FsFn1           rdf:type                fs:FSFileName
FsFn1           fsfn:name               "Root Object/Registry_file"
Fs01            rdf:type                fs:File
Fs01            fs:containsFSFileName   FsFn1
Fs01            fs:childOf              Fs00

Partition1      fs:hasFSFileName        FsFn2
FsFn2           rdf:type                fs:FSFileName
FsFn2           fsfn:name               "Root Object/UserData"
Fs02            rdf:type                fs:Folder
Fs02            fs:containsFSFileName   FsFn2
Fs02            fs:childOf              Fs00

Partition1      fs:hasFSFileName        FsFn3
FsFn3           rdf:type                fs:FSFileName
FsFn3           fsfn:name               "Root Object/System"
Fs03            rdf:type                fs:Folder
Fs03            fs:containsFSFileName   FsFn3
Fs03            fs:childOf              Fs00

Partition1      fs:hasFSFileName        FsFn4
FsFn4           rdf:type                fs:FSFileName
FsFn4           fsfn:name               "Root Object/Programs"
Fs04            rdf:type                fs:Folder
Fs04            fs:containsFSFileName   FsFn4
Fs04            fs:childOf              Fs00

Partition1      fs:hasFSFileName        FsFn5
FsFn5           rdf:type                fs:FSFileName
FsFn5           fsfn:name               "Root Object/UserData/picture1"
Fs05            rdf:type                fs:File
Fs05            fs:containsFSFileName   FsFn5
Fs05            fs:childOf              Fs02
```

```
Partition1      fs:hasFSFileName        FsFn6
FsFn6           rdf:type                fs:FSFileName
FsFn6           fsfn:name               "Root Object/UserData/Malware"
FsO6            rdf:type                fs:File
FsO6            fs:containsFSFileName   FsFn6
FsO6            fs:childOf              FsO2

Partition1      fs:hasFSFileName        FsFn7
FsFn7           rdf:type                fs:FSFileName
FsFn7           fsfn:name               "Root Object/UserData/ImportantDocument"
FsO7            rdf:type                fs:File
FsO7            fs:containsFSFileName   FsFn7
FsO7            fs:childOf              FsO2

Partition1      fs:hasFSFileName        FsFn8
FsFn8           rdf:type                fs:FSFileName
FsFn8           fsfn:name               "Root Object/System/Kernel"
FsO8            rdf:type                fs:File
FsO8            fs:containsFSFileName   FsFn8
FsO8            fs:childOf              FsO3

Partition1      fs:hasFSFileName        FsFn9
FsFn9           rdf:type                fs:FSFileName
FsFn9           fsfn:name               "Root Object/Programs/Browser"
FsO9            rdf:type                fs:File
FsO9            fs:containsFSFileName   FsFn9
FsO9            fs:childOf              FsO4

Partition1      fs:hasFSFileName        FsFn10
FsFn10          rdf:type                fs:FSFileName
FsFn10          fsfn:name               "Root Object/Programs/FileExplorer"
FsO10           rdf:type                fs:File
FsO10           fs:containsFSFileName   FsFn10
FsO10           fs:childOf              FsO4
```

Listing 6.1: Sample hard disk triples

6.10.2 Random Access Memory

Listing 6.2 shows the triples that represent the data from the random access memory. The Process `Malware` is not visible with the standard tools available in the operating system but it can be found because it is stored in the memory.

```
pro:ProcessList         pro:hasProcess          Proc0
Proc0                   pro:parent              Proc0
Proc0                   pro:name                "Kernel"

pro:ProcessList         pro:hasProcess          Proc1
Proc1                   pro:parent              Proc0
Proc1                   pro:name                "Browser"
Proc1                   pro:hasConnection       "www.google.com"

pro:ProcessList         pro:hasProcess          Proc2
Proc2                   pro:parent              Proc0
Proc2                   pro:name                "FileExplorer"
Proc2                   pro:hasResource         FsO5
Proc2                   pro:hasConnection       "www.malicious-server.com"
```

```
pro:ProcessList         pro:hasProcess          Proc3
Proc3                   pro:parent              Proc3
Proc3                   pro:name                "Malware"
```

Listing 6.2: Sample memory triples

6.10.3 Registry

Listing 6.3 shows the triples that represent an excerpt of the data of the registry on the hard disk. The registry data of the memory is not shown as the difference is only the value of the firewall status.

```
H1      rdf:type        reg:Hive
H1      reg:root        K0
H1      reg:name        "Hive1"

K0      rdf:type        reg:Key
K0      reg:name        "Root"
K0      reg:hasSubKey   K1

K1      rdf:type        reg:Key
K1      reg:name        "Firewall"
K1      reg:keystate    S1
K1      reg:hasValue    V1

S1      rdf:type        reg:State
S1      rdf:value       "S"

V1      rdf:type        reg:Value
V1      reg:type        T1
V1      reg:key         "Status"
V1      reg:value       "1"

T1      rdf:type        reg:ValueType
T1      rdf:value       "DWORD"
```

Listing 6.3: Sample registry triples

Chapter 7

Implementation

This chapter presents the parts that were implemented and the difficulties that were overcome. At first an overview of the architecture is outlined and then some details of the implementation and used tools are presented.

7.1 Overview

The first thing that was implemented was the ontology. It consists of the nine files described in chapter 6. Afterwards, a converting tool was written in Java that converts the output of several forensic tools to RDF files that fit to the ontology definitions specified in the RDFS files. Then the RDF files were then automatically loaded to the selected database. In the end several SPARQL queries were developed to find evidence in the database.

7.2 RDFS

The structure of the ontology is written with RDFS. The files were generated with *SemanticWorks* from section 5.2.1 and later edited by hand with a normal text editor. A problem when creating the files was that some tools, that can create RDF files, for example *Protégé* from section 5.2.2, use OWL elements or produce too much unneeded elements. A main problem is that only few tools support importing other files for namespaces as *SematicWorks* does.

The structure of the ontology was chosen to be intuitively comprehensible. It was started from hardware view with the hard disk, the random access memory and the network interface card. Then the software structures were modelled as they can be found in operating systems.

7.3 RDF

The next step was to create the program that converts the output of the forensic tools that extract the information from the hard disk and random access memory images. Most forensic tools print their output to the standard output. The program, which is written in Java, asks for the location of the hard disk and memory images and the database to use and then runs the tools with the correct parameters. The output of the different tools is parsed and converted to RDF files that use the structures of the ontology. These RDF files are then stored in the selected database. The execution of the tools is partly parallelized to speed up the process. A screenshot of the program is shown in the appendix in listing C.4. The information about the tools that are used, the location of the database and other information that is needed to run the program is stored in a configuration file. An example for this file is shown in the appendix in listing A.1. The listing A.2 shows an example for the progress information printed by the program.

One important point when filling the ontology is to add the data with references to the tools it way retrieved by. But if all values, timestamps and tools are added to an object, one cannot distinguish which timestamps and values belong to which tool. So blank nodes are put between the object and the corresponding value to eliminate this indistinguishability. For example, in figure 7.1 there is a registry key for which two tools found different values and the blank nodes "_:blanknode*" helping to separate these.

7.4 Volatility plugin: hivedump2

When implementing the extraction of the registry from the random access memory respectively the image of it, the first solution was it to call the *printkey* function of *volatility* for each hive. The hive addresses can be located with the *hivelist* function. The output of the *printkey* function for one registry key looks similar to listing 7.1.

```
Registry: \Device\HarddiskVolume1\WINDOWS\system32\config\SECURITY
Key name: SECURITY (S)
Last updated: 2012-09-20 10:33:58
Subkeys:
  (S) Policy
  (S) RXACT
  (V) SAM
Values:
```

Listing 7.1: Sample output of printkey

If this function is executed with only the hive specified, it prints the root key of the hive. From this output one can extract the sub keys. For each of the sub keys the *printkey* function is called again. If this is done recursively,

Figure 7.1: Blank node

the complete registry tree is retrieved. The problem about this approach is that the *printkey* function calls take longer time with the depth of the key in the tree. In tests it started with around 30 seconds and went up to over one minute per call. The registry of the test systems contains between 45000 and 50000 keys. It would take approximately $47500 * 45$ seconds $= 2137500$ seconds $\approx 24,7$ days to extract all keys this way. It did not make it much better to parallelize the task as it would take $24,7$ days / $8 \approx 3$ days for eight parallel function calls. The function *hivedump* prints all keys for a specified hive and does this faster than $24,7$ days. But it only prints the last write date and the key and not the sub keys and values.

The solution is to combine the code of both functions to create *hivedump2* which lists all keys like the *hivedump* function but prints the detailed information of the *printkey* function. The new function is executed parallel for all hives the consequence of which is a duration of around two hours for the extraction of the registry.

7.5 Database

At first *Neo4J*[Neo Technology, Inc, 2013] was used for storing the RDF and RDFS tuples. The database has the advantage of a graphical web front end which visualizes the nodes and their connections. When later in the implementation the registry data was added this front end was very slow. This is no big problem as the database only has to answer queries. But the database was also very slow or did not respond at all.

After this problem *Neo4J* was replaced by the *Sesame* framework[Aduna, 2012]. The database of *Sesame* is much faster and allows SPARQL queries from the web front end what makes developing new queries easier. The query that did not respond in *Neo4J* returned a result after 100ms. In the current version of the Java program the user can select which database should be used.

7.6 SPARQL

The creation of queries to retrieve information from the database is exemplified with a query that finds a specific file name, a query that lists all values of the *autorun* keys in the registry and a query that finds processes that have no parent or only themselves. At all listed queries the prefix shown in listing 7.2 is omitted to make them easier to view and to save space.

7.6.1 Find File

A first try is to use the query from listing 7.3. The result should be the `File Name` entry of the specified file. In line one the variable `?file` is spec-

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX usr:<http://www.example.org/user#>
PREFIX hw:<http://www.example.org/hardware#>
PREFIX fs:<http://www.example.org/filesystem#>
PREFIX sw:<http://www.example.org/software#>
PREFIX pro:<http://www.example.org/process#>
PREFIX reg:<http://www.example.org/registry#>
PREFIX for:<http://www.example.org/forensic#>
PREFIX net:<http://www.example.org/network#>
PREFIX mem:<http://www.example.org/memory#>
PREFIX cnt:<http://www.w3.org/2011/content#>
PREFIX fsfn:<http://www.example.org/filesystem/fsfilename#>
PREFIX base:<http://www.example.org/>
```

Listing 7.2: SPARQL Prefix

ified to be the result. The `DISTINCT` keyword eliminates duplicates in the result set. Line three tells that the object that is bound to the variable `?file` has to be of type `fs:FSFileName`. In line four the value of the `fsfn:name` property of the variable `?file` is required to be the same as `FILENAME`.

```
1 SELECT DISTINCT ?file
2  WHERE {
3  ?file rdf:type fs:FSFileName .
4  ?file fsfn:name "FILENAME".
5 }
```

Listing 7.3: Simple Find file Query

A problem with this simple query is that the full path of the file has to be specified in order to find the FileName entry.

An advanced query that solves this problem is shown in listing 7.4. It uses regular expressions to find files where only parts of the name are known. The query will list all files that contain the specified string in the full path. What we want to get is the name of a file and a reference to the related `File Name` entry for further inspection of the properties of the file. The variables `?name` and `?file` will contain this information. In line three the variable `?file` is ensured to be of the type `fs:FSFileName`. Line four binds the value of the `fsfn:name` field of the variable `?file` to the variable `?name`. The fifth line filters for the specified file name(`FILENAME`). The function `str(a)` returns the string representation of the variable `a` and `regex(a,b)` returns true if string `a` matches pattern `b`. The `regex(a,b,f)` function accepts additional flags. For example flag `i` causes case insensitive matching so `regex("WiNdOwS","windows","i")` returns true.

```
1 SELECT DISTINCT ?name ?file
2  WHERE {
3  ?file rdf:type fs:FSFileName .
4  ?file fsfn:name ?name .
5  FILTER(regex(str(?name), "FILENAME"))
6 }
```

Listing 7.4: Find file Query

7.6.2 Autorun

According to [Microsoft, 2010] the entries for the *Run* and *RunOnce* keys are located in the paths shown in listing 7.5.

```
HKEY_LOCAL_MACHINE/Software/Microsoft/Windows/CurrentVersion/Run
HKEY_CURRENT_USER/Software/Microsoft/Windows/CurrentVersion/Run
HKEY_LOCAL_MACHINE/Software/Microsoft/Windows/CurrentVersion/RunOnce
HKEY_CURRENT_USER/Software/Microsoft/Windows/CurrentVersion/RunOnce
```

Listing 7.5: Autorun registry paths

A common part of these paths is `Windows/CurrentVersion/Run`. The goal of the query is to retrieve the key-value pairs that are the values of these keys. The resulting query is shown in listing 7.6. At first, in lines three and four a key that has the name `Windows` is selected and bound to the variable `?win`. Line five binds a sub key of the one in `?win` to the variable `?cv`. The lines six and seven ensure that the key in `?cv` has the name `CurrentVersion`. Line eight works similarly to line five and binds the subkeys of `?cv` to `?run`. Line ten filters the value of `?run` to contain `Run` in its name. This way `Run`, `RunOnce` and all other keys that contain run like `RunServices` and `RunServicesOnce` are included. The lines eleven to thirteen extract the key-value pairs to the variables `?name` and `?command`.

7.6.3 Parent Process

Some malware tries to hide by removing itself from the process hierarchy that starts with one process. This can be detected by looking at the line of ancestors of each process. If one process is its own parent or does not originate from the most basic process, it might have tried to hide. Of course the most basic parent process is always in the result set as it was specified to be its own parent in section 6.5. For this query again two possibilities exist. The first one is shown in listing 7.7. Lines three and four bind a `pro:Process` object to the variable `?child` that is the process the query will examine. The `OPTIONAL` keyword specifies that the restrictions in the following block, that is indicated by braces, do not necessarily need to match. The `*` in line six

```
1  SELECT DISTINCT ?nrun ?name ?command
2  WHERE {
3   ?win rdf:type reg:Key .
4   ?win reg:name [ rdf:value "Windows" ] .
5   ?win reg:hasSubKey [ rdf:value ?cv ] .
6   ?cv rdf:type reg:Key .
7   ?cv reg:name [ rdf:value "CurrentVersion" ] .
8   ?cv reg:hasSubKey [ rdf:value ?run ] .
9   ?run reg:name [ rdf:value ?nrun ] .
10  FILTER(regex(str(?nrun), "Run", "i")) .
11  ?run reg:hasValue [ rdf:value ?value ] .
12  ?value reg:key [ rdf:value ?name ] .
13  ?value reg:value [ cnt:ContentAsText ?command ] .
14  }
```

Listing 7.6: Autorun Query

says that the `pro:parent` predicate can be matched not, once or multiple times. So the variable `?parent` is bound to the one of the ancestors of `?child` regarding the `pro:parent` relation. The structure of the process part of the ontology requires that the most basic parent process has itself as parent as demanded in section 6.5. Along with line seven this leads to the fact that the `?parent` variable must be the most basic root. The filter in line eight ensures that the predicate in line six is not applied zero times. If any of the restrictions in the `OPTIONAL` block does not match the variable `?parent` is not bound. This case is filtered in line ten, so only those processes are included in the result that do not have a parent according to the restrictions in the `OPTIONAL` block.

```
1  SELECT DISTINCT ?parent ?child ?childname
2  WHERE {
3   ?child rdf:type pro:Process .
4   ?child pro:Name [rdf:value ?childname] .
5   OPTIONAL{
6     ?child pro:parent* ?parent .
7     ?parent pro:parent ?parent .
8     FILTER(?child != ?parent)
9   }
10  FILTER(!BOUND(?parent))
11 }
```

Listing 7.7: Parent Process SELECT query

Another possibility to achieve the result is to use the query from listing 7.8. A `CONSTRUCT` query returns a RDF graph.

The first line specifies that the triples in the result graph are built using the variables `?s`, `?p` and `?o`. The `UNION` keyword specifies that one of the two restriction blocks before or after the word needs to match.

The first block defines that `?s` and `?o` must be of the type `pro:Process`, that `?o` is the parent process of `?s`. Line eight says that for this block the predicate of the result triple is a relation between `?s` and `?o`.

Line twelve specifies that `?s` is of the type `pro:Process`. The next two lines bind `?o` to the name of the process in `?s` and `?p` to a relation between `?s` and `?o`. The filter in line fifteen is needed because there is more than one possible binding for `?p`. This is not needed in the other block because there is no other connection between two processes. If the query is issued in the *Sesame* web front end, the resulting triples are shown and the RDF graph can be downloaded. The downloaded file can then be analysed in other tools like *RDF Gravity*. An example graph that contains malicious processes that tried to hide by removing the connection to their parent process and is being visualized with *RDF Gravity*, is shown in the appendix in figure C.2. The processes can be found in the upper left corner of the drawing layer.

```
1  CONSTRUCT { ?s ?p ?o . }
2  WHERE
3  {
4   {
5    ?s rdf:type pro:Process .
6    ?o rdf:type pro:Process .
7    ?s pro:parent ?o .
8    ?s ?p ?o .
9   }
10  UNION
11  {
12   ?s rdf:type pro:Process .
13   ?s pro:Name [ rdf:value ?o ] .
14   ?s ?p [ rdf:value ?o ] .
15   FILTER(regex(str(?p),str(pro:Name)))
16  }
17 }
```

Listing 7.8: Parent Process CONSTRUCT query

7.7 Adding additional data: Log files

As mentioned in section 4.2.3.3 one might want to have additional data in the ontology. As an example log files are added.

At first a new RDFS file is created which describes the data in the logs and specifies the new type `log:LogFile`. In the specification a property is necessary that connects the log file to the associated `fs:FileSystemObject`. Next it may contain a list of log entries. The kind of data these entries must contain depends on the kind of log files that are considered. The resulting structure may look like figure 7.2. To be able to find all log files a property

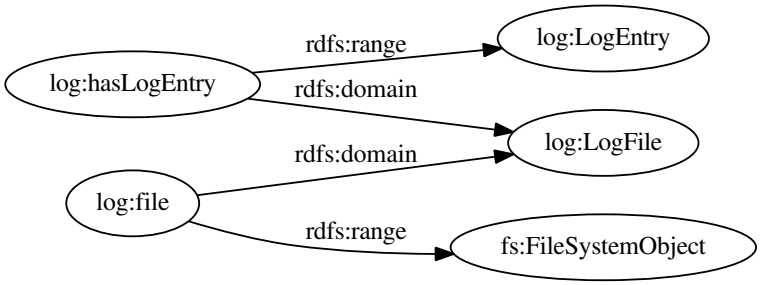

Figure 7.2: Example for Log.rdfs

`sw:hasLogFile` with `rdf:range log:LogFile` and `rdf:domain sw:Kernel` is added to `Software.rdfs`. The new `Software.rdfs` looks similar to figure 7.3.

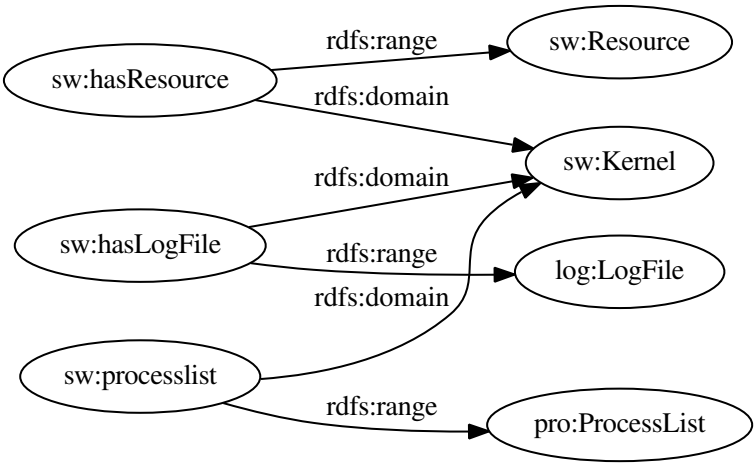

Figure 7.3: Software.rdfs with LogFile

7.8 Statistics

- The ontology contains 55 classes and 58 properties.
- The program that asks for the necessary information, executes the forensic programs, converts the output, stores the converted output in the selected database and allows to query the database consists of 4470 lines of Java code.
- The input data was 10 GB hard disk and 256 MB random access memory.
- The extracted data, split up to several RDF files, has a size of around 460 MB for each case.
- The extraction process takes around two hours on a Intel Core i7 CPU Q 820 with 1.73GHz.
- The Sesame database for each case has a size of around 700 MB.

Chapter 8

Evaluation

This chapter shows the effectiveness and efficiency of the developed procedure by applying it to four cases. The cases consist of real malware samples that are run on a test system and analysed later.

8.1 Procedure

A *VirtualBox*[Oracle, 2012] virtual machine with *Windows XP SP3* was installed and a snapshot was taken at the first start. The virtual machine has 10 GB hard disk space and 256 MB random access memory. *VirtualBox* is started with `--dbg --startvm <VM name>` parameters to be able to use the debug console.

For each malware sample the following steps were taken:

1. At first the malware was inserted with a virtual CD, started and allowed to run for a while. Depending on the malware, the system was restarted to find traces that make the malware run at every start of the operating system.

2. Next the system was paused.

3. To extract the hard disk, the command
   `VBoxManage clonehd <infile> <outfile> --format RAW`
   was used from a command line.

4. To extract the random access memory, the command

   `.pgmphystofile <outfile>`

   was executed in the debug console that can be started by the `Command line...` button in the `Debug` menu.

5. For the next step, the developed program was started to extract the information from the snapshots and put it into the database. As explained in section 7.8, this step took around two hours.

6. The last step, was to query the database for evidence. This step is described in sections 8.3, 8.4, 8.5 and 8.6. For these sections it is assumed that it is not known what kind of malware was run. The approach is to run selected queries to get an overview and then examine more in detail.

8.2 SPARQL Queries

This section shows the queries that are used to find traces of malware. Some of them are already explained in detail in chapter 7.

8.2.1 Find file

A query to find files as shown in listing 7.4 is explained in section 7.6.1. `FILENAME` has to be replaced by the string to search for. The result contains all files that have the given string in their path.

8.2.2 Autorun

The query shown in listing 7.6 is explained in section 7.6.2. It lists all key-value pairs of the values of the `Windows/CurrentVersion/Run` registry subtrees. These subtrees contain the programs that are started with the operating system.

8.2.3 Network

Another sign for malware may be the network connections. The query from listing 8.1 can be used to find all processes that have TCP connections. This returns the processes and what they are connected to.

Replace the two tools in the `hasForensicTool` lines by their socket equivalent (see 4.3.2) to search for all network protocols.

8.2.4 Parent Process

The query explained in section 7.6.3 and shown in listing 7.7 lists all processes that do not have a normal line of ancestors.

8.2.5 Resources

A useful piece of evidence is which resources a process uses. To obtain this information the query from listing 8.2 can be used. The query binds the process to the variable `?pid` and filters the name matching the regular expression `PROCESSNAME`. If the resource is a file, the `OPTIONAL` block binds the name of the file to the variable `?filename`.

```
SELECT DISTINCT ?pid ?name ?conn ?local ?remote
WHERE {
  {?pid for:hasForensicTool base:volatility_connscan . }
 UNION
  {?pid for:hasForensicTool base:volatility_connections . }
 ?pid rdf:type pro:Process .
 ?pid pro:Name [rdf:value ?name] .
 ?pid pro:hasConnection ?conn .
 ?conn pro:Local_Address [rdf:value ?local] .
 ?conn pro:Remote_Address [rdf:value ?remote] .
}
```

Listing 8.1: Processes with connections

```
SELECT DISTINCT ?pid ?name ?resource ?filename
WHERE
{
 ?pid rdf:type pro:Process .
 ?pid pro:Name [rdf:value ?name] .
 FILTER(regex(str(?name), "PROCESSNAME", "i"))
 ?pid pro:hasResource ?resource .
 OPTIONAL{ ?resource fsfn:name ?filename . }
}
```

Listing 8.2: Resources of a process

8.3 Case 1

The Autorun query from section 8.2.2 returns `key: "mydnswatch.exe" value:"C:\mydnswatch\mydnswatch.exe"` besides the normal values. A lookup on the internet for `mydnswatch.exe` proves that malware was running on the computer. The first query from section 8.2.3 shows that the processes `explorer.exe` and `winlogon.exe` have connections. Both of them should not have any connection. All addresses the both processes are connected to do not look familiar and a search on the internet shows that they belong to malware related servers.

8.4 Case 2

The Autorun query from section 8.2.2 returns `key: "CTEMON.EXE" value: ""C:\Dokumente und Einstellungen\All Users\Application Data\winlogon .exe" /h"` besides the normal values. The first thing that confuses is that

key (`CTEMON.EXE`) value (`winlogon.exe`) do not match. The next clue is that `winlogon.exe` does not belong to that folder.

The Connection query from section 8.2.3 returns no connections.

If the Find File query from section 8.2.1 is executed for `winlogon.exe`, it is not found at the given location. A lookup on the internet for `CTEMON.EXE` proves that malware was running on the computer.

8.5 Case 3

For the third malware sample the Autorun query from section 8.2.2 returns only the normal values. The Connection query from section 8.2.3 returns a process that is named `pdvdbny.exe` and has two connections to unknown addresses. The Find File query from section 8.2.1 shows that the file is located at `/Dokumente und Einstellungen/Administrator/Lokale Einstellungen/Anwendungsdaten/pdvdbny.exe`. Usually, programs are not located at this place. The Parent Process query from section 8.2.4 shows that the processes `pdvdbny.exe`, `ctfmon.exe` and `explorer.exe` are detached from their relation to the `Init` process. A graphical view of the result of the alternative version of the Parent Process query is shown in the appendix in figure C.2. The Resources query from section 8.2.5 shows that the process accesses the files that store cookies, temporary internet files and the history. Additionally it accesses the two files `/WINDOWS/WinSxS/x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83` and `/WINDOWS/WinSxS/x86_Microsoft.Windows.GdiPlus_6595b64144ccf1df_1.0.2600.5512_x-ww_dfb54e0c`. It can access and edit the data stores of the browser, so it can manipulate the websites a user accesses or steal sessions. This program is definitely up to no good.

8.6 Case 4

The Autorun query from section 8.2.2 returns only the normal values. In contrast the Connection query from section 8.2.3 returns a process named `securedoc.html.` that has two connections to unknown addresses. The Parent Process query shows similar to section 8.5 that the processes `securedoc.html.`, `ctfmon.exe` and `explorer.exe` are detached from their relation to the `Init` process. And except the last file the process accesses the same files as the one in section 8.5.

Chapter 9

Summary

The goal of this work has been defined as developing an ontology that makes it easier to investigate security incidents. The required features include automatic extraction of evidence from a computer and a way to gather evidence from the extracted data.

The ontology was created, using existing generic structures for the different forensically interesting parts of the computer. In this work the hard disk and the random access memory were dealt with. To accomplish the extraction of the data from these sources, existing forensic tools are used and their output is converted to a format that conforms with the structure of the ontology. To gather evidence from the converted data, it is stored in a queryable triple store. A collection of queries has been developed and tested on real malware samples (section 7.6 and chapter 8).

A problem with the visualization of ontologies is that in contrast to other markup language, for example UML, the graphical representation of the individual elements is not specified.

What cannot yet be found by the ontology are traces that are located in the content of the files on the hard disk or in the code respectively the data of the random access memory. Such information can be integrated in the developed ontology by including additional forensic tools. Any other information that may be useful to investigate a case can be included in the ontology as it is shown in section 7.7.

An advantage of the XML based structure of RDF is the easiness of generating it from the output of different tools.

That the provided queries can be used to find traces of malware was shown in chapter 8.

Traces of malware can also be found by virus scanning programs. But in contrast to these, the ontological approach allows to find malware for which there does not yet exist a signature or behaviour profile or any other characteristic for detection. Additionally it is not the purpose of this approach to compete with such programs, as forensic analysis most times takes place

after some issue occurred.

Using the ontology to investigate the cases made it easier because only queries had to be issued. For example, for the Autorun query the *printkey* module of *volatility* needs to be run multiple times to get the data from the registry in the memory. For the registry on the hard disk it is first needed to know where the relevant files are located and then extract them with *icat* from *The Sleuth Kit* if the source is an image and then run *reglookup* for each of them. All this is simplified to only issuing one query because all other commands have been run automatically during the extraction process.

In my opinion the ontological approach has great potential because it makes forensic analysis easier. An additional aspect is that after the database is filled, multiple investigators can use this data and do not need to run the same tools again.

The creation of the ontology is not that easy as the tools do not create satisfying RDFS files or are uncomfortable to use. If the forensic ontology is to be developed further, it may be useful to create a tool that allows an easier creation of RDFS files of similar structure. Additionally, a program should be developed that makes it easier to parse the output of a forensic tool and map it to the correct part of the ontology.

Appendix A

Extraction tool listings

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>ExtractInformation-Config-File</comment>
<entry key="samdump2">/usr/bin/samdump2</entry>
<entry key="started">1353417866724</entry>
<entry key="sesamerepo">case1</entry>
<entry key="ramimage">../cases/case1/zeus.vmem</entry>
<entry key="fsstat">../TSK/fsstat</entry>
<entry key="hddimage">../cases/case1/zeus.raw</entry>
<entry key="threadcount">10</entry>
<entry key="skipregistry">true</entry>
<entry key="reglookup">../reglookup/reglookup</entry>
<entry key="ontology">../ONTO</entry>
<entry key="database">sesame</entry>
<entry key="datadir">../cases/case1/extracted</entry>
<entry key="bkhive">/usr/bin/bkhive</entry>
<entry key="python">/usr/bin/python</entry>
<entry key="fls">../TSK/fls</entry>
<entry key="hddimageoffset">63</entry>
<entry key="baseuri">http://www.0x221b.org/</entry>
<entry key="mmls">../TSK/mmls</entry>
<entry key="sesameserver">http://..:8080/openrdf-sesame/</entry>
<entry key="volatility">../volatility-2.1/vol.py</entry>
<entry key="icat">../TSK/icat</entry>
<entry key="skipprocesses">true</entry>
</properties>
```

Listing A.1: Configuration of the tool(shortened)

```
Start
Start loading Ontology
Start writing User.rdfs (2.0KB)
Finished writing User.rdfs
Start writing Filesystem.rdfs (8.0KB)
Finished writing Filesystem.rdfs
Start writing Forensic.rdfs (1.0KB)
Finished writing Forensic.rdfs
Start writing Registry.rdfs (4.0KB)
Finished writing Registry.rdfs
Start writing Process.rdfs (2.0KB)
Finished writing Process.rdfs
Start writing Hardware.rdfs (1.0KB)
```

```
Finished writing Hardware.rdfs
Start writing Memory.rdfs (7.0KB)
Finished writing Memory.rdfs
Start writing Network.rdfs (4.0KB)
Finished writing Network.rdfs
Start writing Software.rdfs (1.0KB)
Finished writing Software.rdfs
Finished loading Ontology
Start NTFS 2 RDF
Finished NTFS 2 RDF
Start checking files 1/3
Checking file generated_harddisk.rdf
OK
Finished checking files 1/3
Start writing to DB 1/3
Start writing generated_harddisk.rdf (8.6650390625MB)
Finished writing generated_harddisk.rdf
Finished writing to DB 1/3
Start MEM 2 RDF
Finished MEM 2 RDF
Start checking files 2/3
Checking file generated_processes.rdf
OK
Checking file generated_Registry_MEM.rdf
OK
.
.
.
OK
Finished checking files 2/3
Start writing to DB 2/3
Start writing generated_processes.rdf (8.369140625MB)
Finished writing generated_processes.rdf
Start writing generated_Registry_MEM.rdf (0.0KB)
Finished writing generated_Registry_MEM.rdf
.
.
.
Finished writing to DB 2/3
Start Registry 2 RDF
Finished Registry 2 RDF
Start Users
Finished Users
Start Additional Edges
Finished Additional Edges
Start checking files 3/3
.
.
.
Checking file _WINDOWS_system32_config_software_hdd.rdf
OK
Checking file _WINDOWS_system32_config_SECURITY_hdd.rdf
OK
Checking file _WINDOWS_system32_config_default_hdd.rdf
OK
Checking file _WINDOWS_system32_config_SAM_hdd.rdf
OK
Checking file _WINDOWS_system32_config_system_hdd.rdf
OK
Checking file Users.rdf
OK
Checking file additional_edges.rdf
```

```
OK
Finished checking files 3/3
Start writing to DB 3/3
.
.
.
Start writing _WINDOWS_system32_config_software_hdd.rdf (169.330078125MB)
Finished writing _WINDOWS_system32_config_software_hdd.rdf
Start writing _WINDOWS_system32_config_SECURITY_hdd.rdf (710.0KB)
Finished writing _WINDOWS_system32_config_SECURITY_hdd.rdf
Start writing _WINDOWS_system32_config_default_hdd.rdf (4.6513671875MB)
Finished writing _WINDOWS_system32_config_default_hdd.rdf
Start writing _WINDOWS_system32_config_SAM_hdd.rdf (259.0KB)
Finished writing _WINDOWS_system32_config_SAM_hdd.rdf
Start writing _WINDOWS_system32_config_system_hdd.rdf (41.7158203125MB)
Finished writing _WINDOWS_system32_config_system_hdd.rdf
Start writing Users.rdf (2.0KB)
Finished writing Users.rdf
Start writing additional_edges.rdf (159.0KB)
Finished writing additional_edges.rdf
Finished writing to DB 3/3
Finished
Database is at http://localhost:8080/openrdf-sesame/case1
```

Listing A.2: Output of the Process

Appendix B

Forensic tools output listings

```
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: NTFS
Volume Serial Number: BE8CB9D38CB98685
OEM Name: NTFS
Version: Windows XP

METADATA INFORMATION
--------------------------------------------
First Cluster of MFT: 786432
First Cluster of MFT Mirror: 1309293
Size of MFT Entries: 1024 bytes
Size of Index Records: 4096 bytes
Range: 0 - 10576
Root Directory: 5

CONTENT INFORMATION
--------------------------------------------
Sector Size: 512
Cluster Size: 4096
Total Cluster Range: 0 - 2618586
Total Sector Range: 0 - 20948695

$AttrDef Attribute Values:
$STANDARD_INFORMATION (16)   Size: 48-72   Flags: Resident
$ATTRIBUTE_LIST (32)   Size: No Limit   Flags: Non-resident
$FILE_NAME (48)   Size: 68-578   Flags: Resident,Index
$OBJECT_ID (64)   Size: 0-256   Flags: Resident
$SECURITY_DESCRIPTOR (80)   Size: No Limit   Flags: Non-resident
$VOLUME_NAME (96)   Size: 2-256   Flags: Resident
$VOLUME_INFORMATION (112)   Size: 12-12   Flags: Resident
$DATA (128)   Size: No Limit   Flags:
$INDEX_ROOT (144)   Size: No Limit   Flags: Resident
$INDEX_ALLOCATION (160)   Size: No Limit   Flags: Non-resident
$BITMAP (176)   Size: No Limit   Flags: Non-resident
$REPARSE_POINT (192)   Size: 0-16384   Flags: Non-resident
$EA_INFORMATION (208)   Size: 8-8   Flags: Resident
```

```
$EA (224)    Size: 0-65536    Flags:
$LOGGED_UTILITY_STREAM (256)    Size: 0-65536    Flags: Non-resident
```

Listing B.1: Output of fsstat from the sleuth kit

```
Virtual    Physical   Name
---------- ---------- ----
0xe18e7a38 0x09433a38 \??\C:\Do..en\Benutzer1\Lok..en\Anw..en\Mi..\Wi..\UsrClass.dat
0xe18e08d8 0x091a38d8 \Dev..e1\Doku..ellungen\Benutzer1\NTUSER.DAT
0xe156ab60 0x068ceb60 \Dev..e1\Doku..ngen\Lo..\Lok..en\Anw..en\Mi..\Wi..\UsrClass.dat
0xe1561ac8 0x068b8ac8 \Dev..e1\Do..en\Lo..\NTUSER.DAT
0xe153d9f0 0x062a89f0 \Dev..e1\Do..en\Net..\Lok..en\Anw..en\Mi..\Wi..\UsrClass.dat
0xe1534b60 0x06213b60 \Dev..e1\Doku..ellungen\NetworkService\NTUSER.DAT
0xe1371218 0x037da218 \Device\HarddiskVolume1\WINDOWS\system32\config\software
0xe1378008 0x04149008 \Device\HarddiskVolume1\WINDOWS\system32\config\SECURITY
0xe1378758 0x04149758 \Device\HarddiskVolume1\WINDOWS\system32\config\default
0xe134ab30 0x03003b30 \Device\HarddiskVolume1\WINDOWS\system32\config\SAM
0xe1254130 0x02269130 [no name]
0xe1018258 0x0202b258 \Device\HarddiskVolume1\WINDOWS\system32\config\system
0xe1007260 0x01feb260 [no name]
0x8068f9bc 0x0068f9bc [no name]
```

Listing B.2: Output of the hivelist module of volatility(shortened)

Appendix C

Screenshots

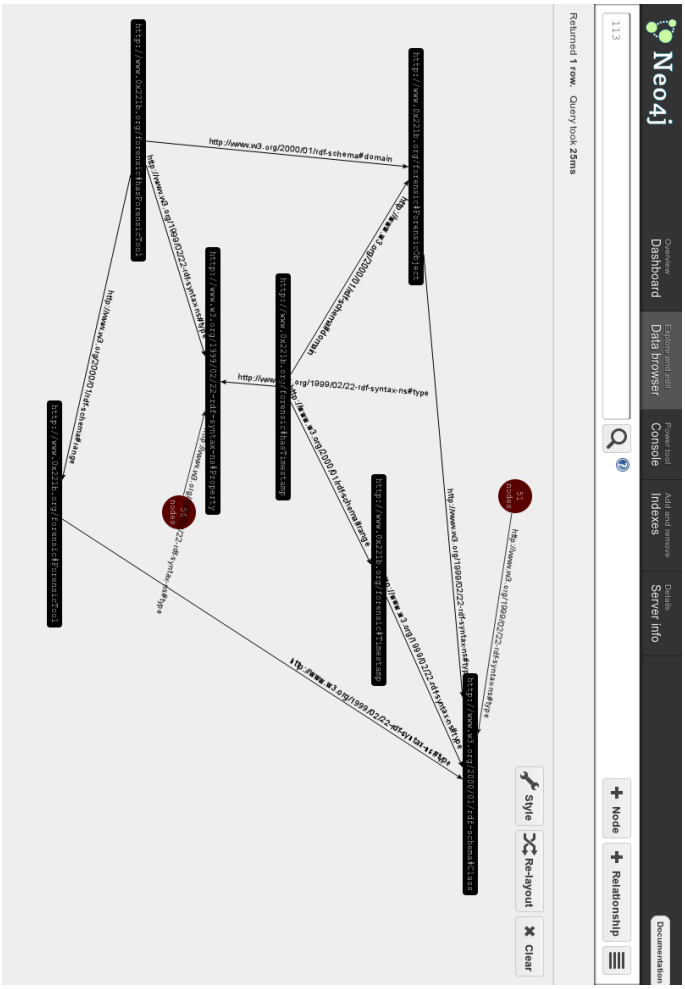

Figure C.1: Neo4J web interface: Interactive graph explorer

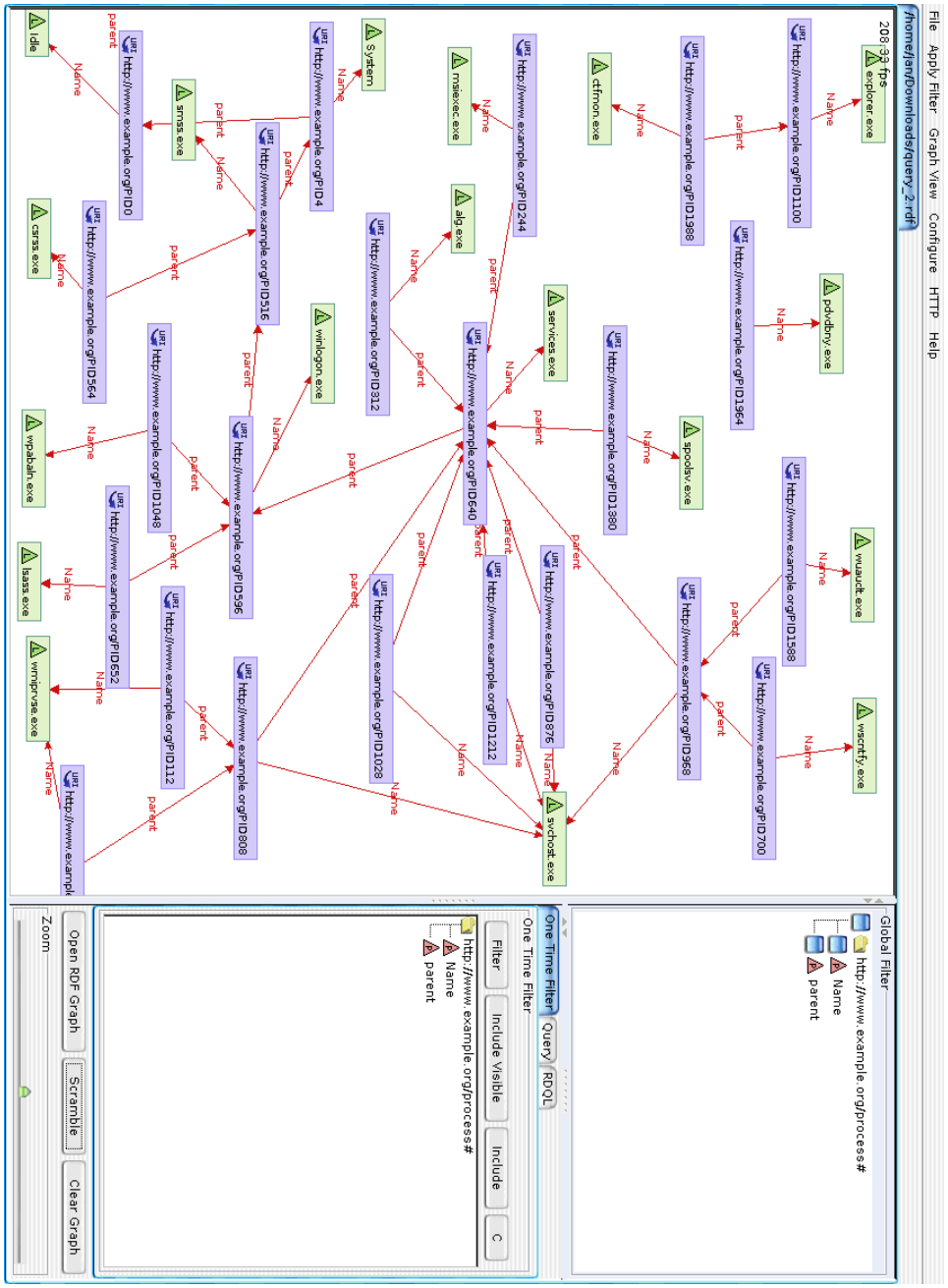

Figure C.2: Gravity interface

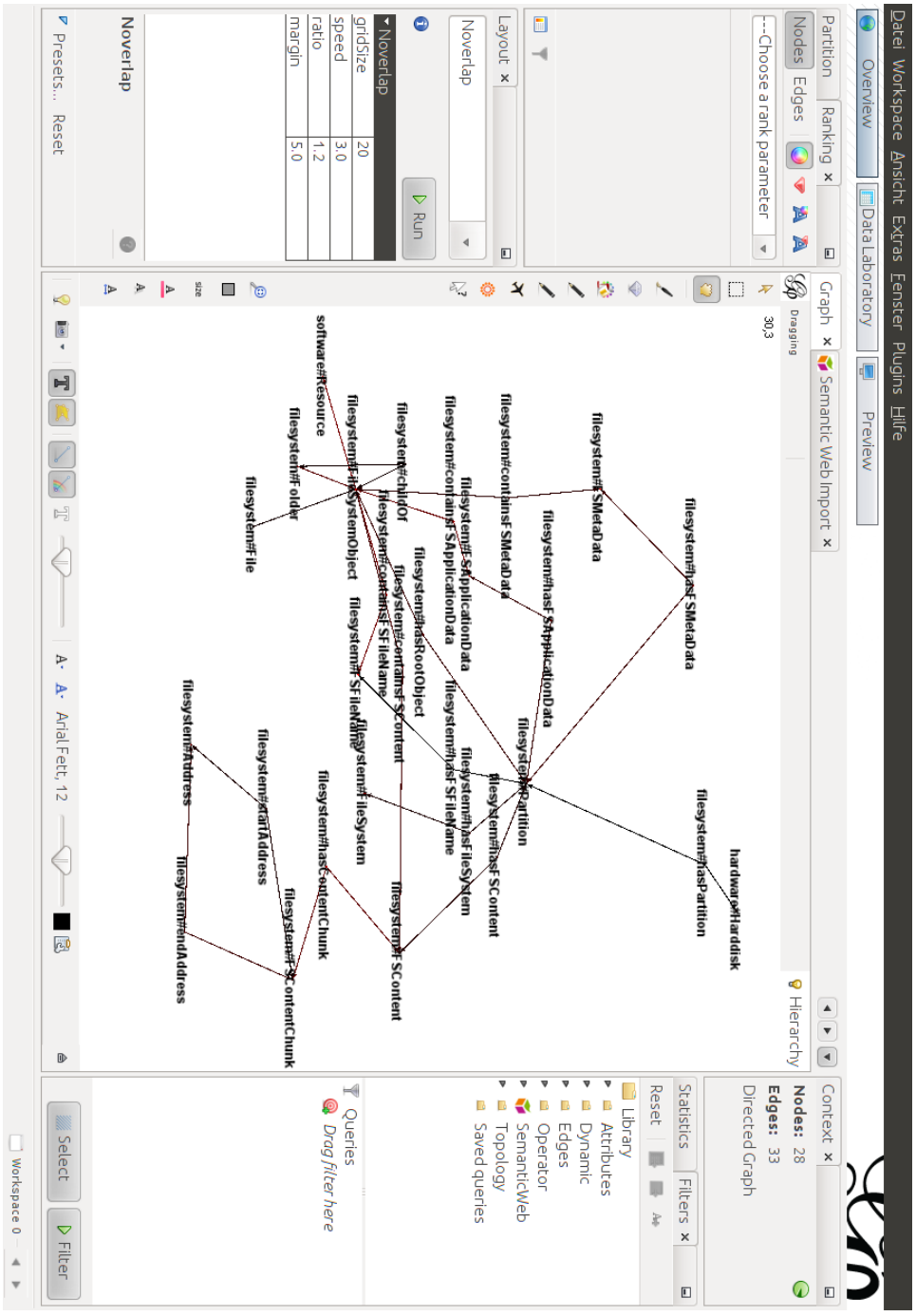

Figure C.3: Gephi interface

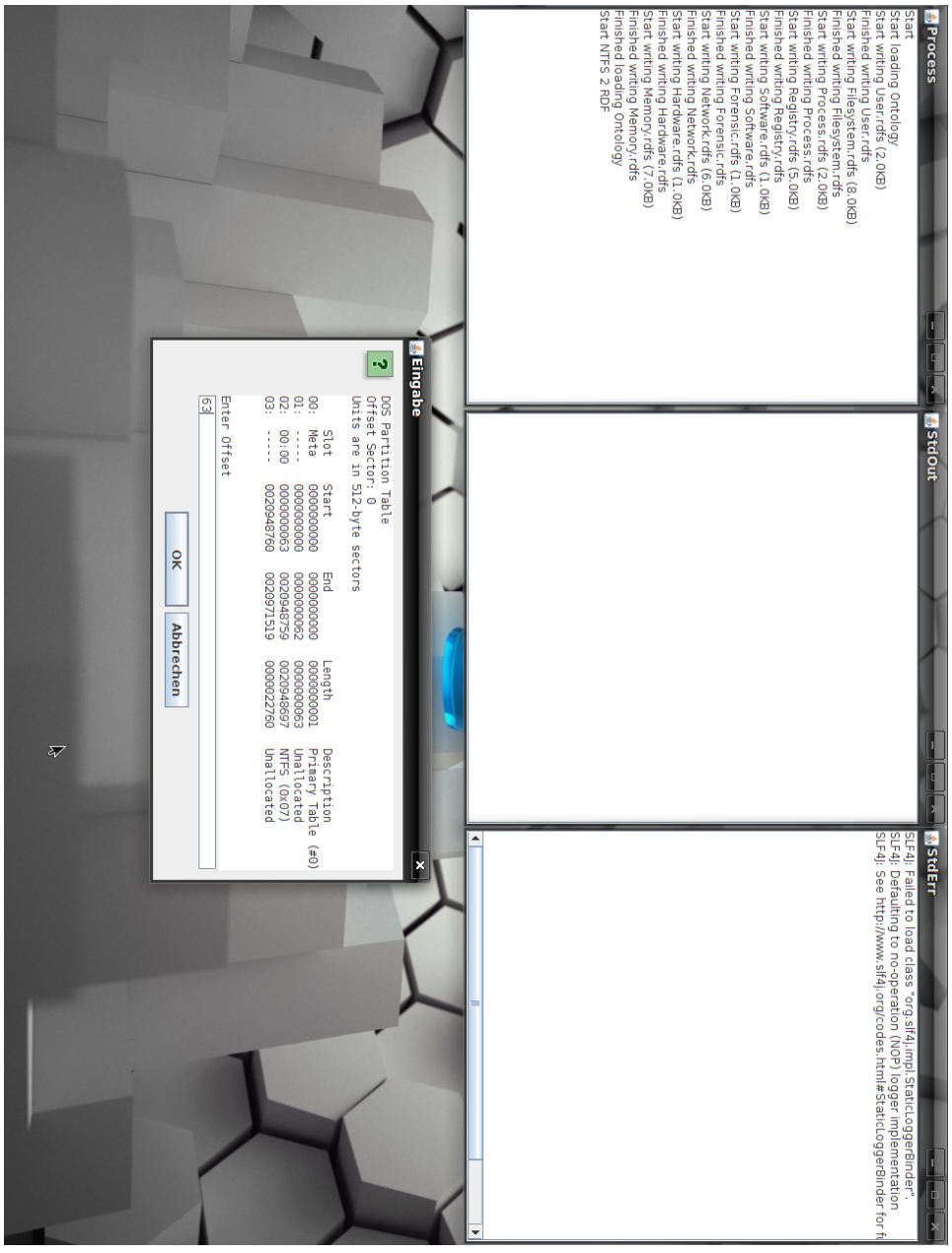

Figure C.4: Extraction tool

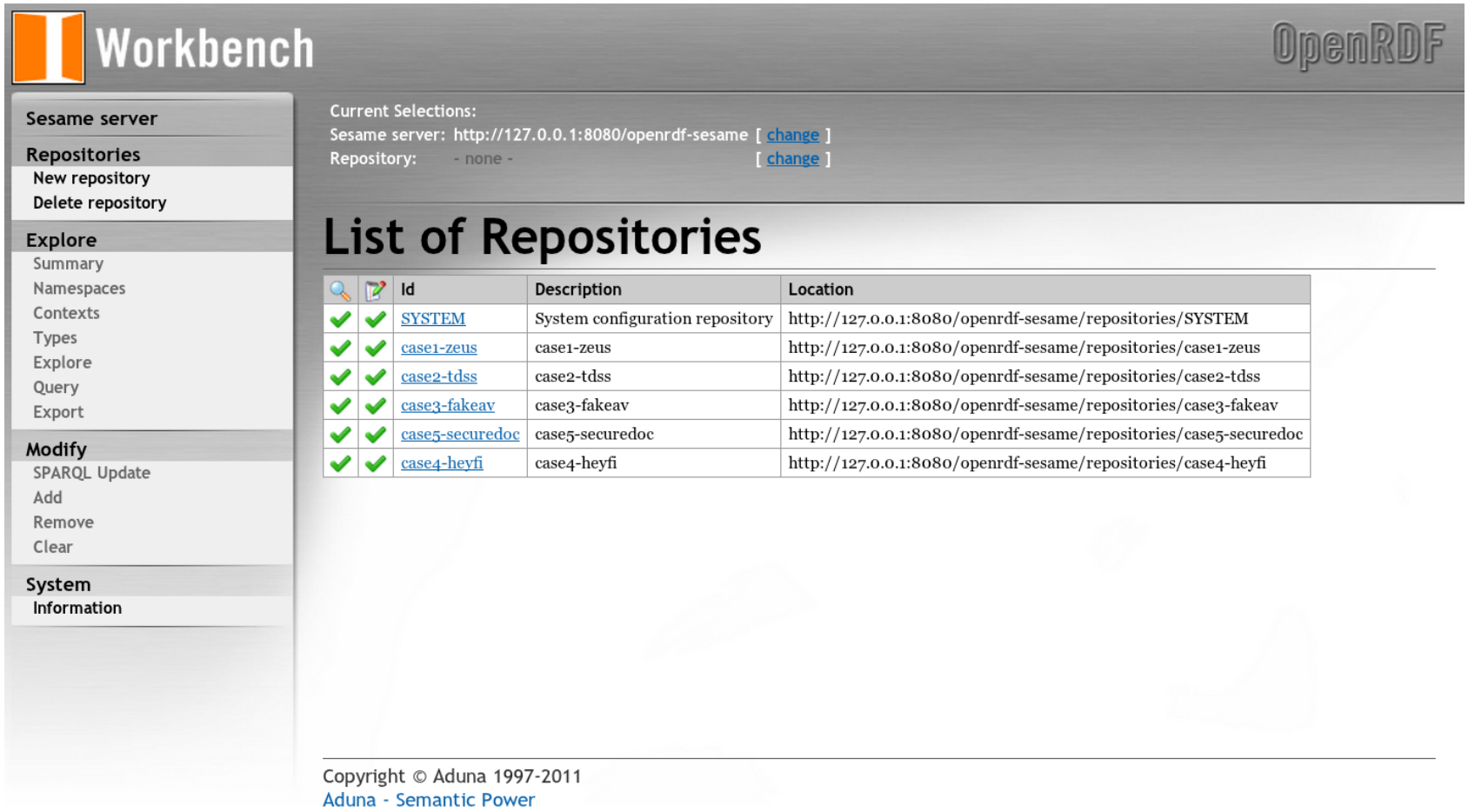

Figure C.5: Sesame web interface: Repository selection

Figure C.6: Sesame web interface: Query input

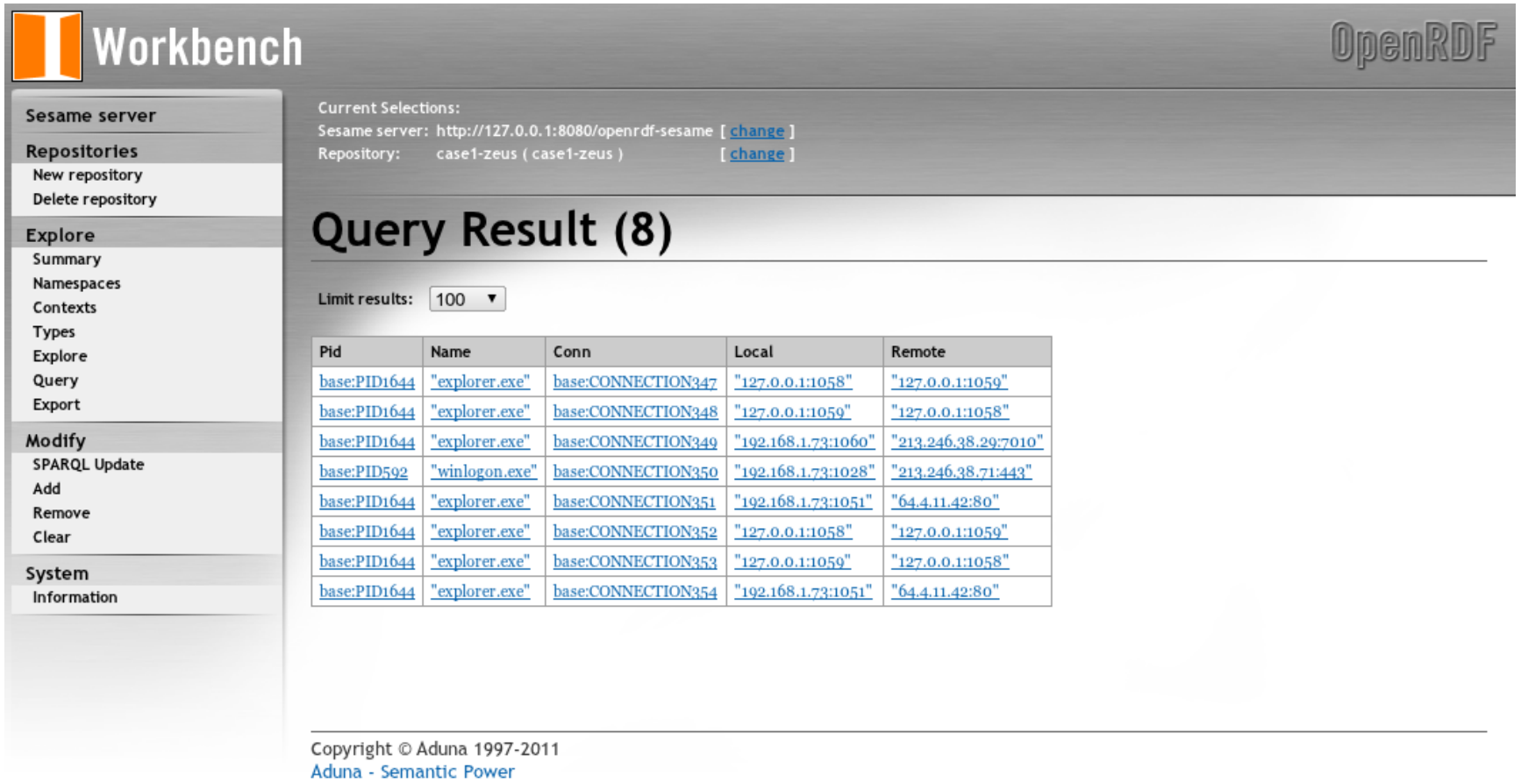

Figure C.7: Sesame web interface: Query result

Bibliography

[Adelstein, 2006] Adelstein, F. (2006). Live forensics: diagnosing your system without killing it first. Commun. ACM, 49(2):63–66. 1

[Aduna, 2012] Aduna (2012). openrdf.org: Home. http://www.openrdf.org/. [last accessed:17.02.2013]. 5.3.2, 7.5

[Altova, 2013] Altova (2013). Altova semanticworks - visual rdf and owl editor that autogenerates rdf/xml and n-triples. http://www.altova.com/products/semanticworks/rdf_owl_editor.html. [last accessed:17.02.2013]. 5.2.1

[Beckett, 2013] Beckett, D. (2013). Raptor rdf syntax library. http://librdf.org/raptor/. [last accessed:17.02.2013]. 5.2.7

[Broekstra et al., 2002] Broekstra, J., Kampman, A., and Harmelen, F. v. (2002). Sesame: A generic architecture for storing and querying rdf and rdf schema. In Proceedings of the First International Semantic Web Conference on The Semantic Web, ISWC '02, pages 54–68, London, UK, UK. Springer-Verlag. 5.3.2

[Carrier, 2012a] Carrier, B. (2012a). The sleuth kit. http://www.sleuthkit.org/sleuthkit/. [last accessed:17.02.2013]. 2, 4.3.1

[Carrier, 2003] Carrier, B. D. (2003). Open source digital forensics tools. @stake Research Report. 4.1.2, 4.1.2, 4.3

[Carrier, 2005] Carrier, B. D. (2005). File System Forensic Analysis. Addison-Wesley, 1 edition. 2, 4.2.1, 4.3.1

[Carrier, 2009] Carrier, B. D. (2009). Body file. http://wiki.sleuthkit.org/index.php?title=Body_file. [last accessed:17.02.2013]. 4.3.1

[Carrier, 2012b] Carrier, B. D. (2012b). The sleuth kit: File and volume system analysis. http://www.sleuthkit.org/sleuthkit/desc.php. [last accessed:17.02.2013]. 4.3.1

[Carrier, 2012c] Carrier, B. D. (2012c). Tsk tool overview - sleuthkitwiki. http://wiki.sleuthkit.org/index.php?title=TSK_Tool_Overview. [last accessed:17.02.2013]. 4.3.1

[Carrier and Grand, 2004] Carrier, B. D. and Grand, J. (2004). A hardware-based memory acquisition procedure for digital investigations. Digit. Investig., 1(1):50–60. 4.1.2.2

[Cohen, 2012] Cohen, M. (2012). Pyflag. http://sourceforge.net/projects/pyflag/, http://www.forensicswiki.org/wiki/PyFlag. [last accessed:17.02.2013]. 2

[Council and Institute, 1998] Council, I. T. I. and Institute, A. N. S. (1998). American National Standard for Information Technology: AT Attachment-3 Interface for (ATA-3). American National Standards Institute. 4.1.2.1

[Cytoscape Consortium, 2012] Cytoscape Consortium (2012). Cytoscape: An open source platform for complex network analysis and visualization. http://www.cytoscape.org/. [last accessed:17.02.2013]. 5.2.5

[Davis, 2008] Davis, N. (2008). Live memory acquisition for windows operating systems: Tools and techniques for analysis. Technical report, Eastern Michigan University. 4.1.2.2

[De Smet, 2009] De Smet, P. (2009). Semi-automatic forensic reconstruction of ripped-up documents. In Document Analysis and Recognition, 2009. ICDAR '09. 10th International Conference on, pages 703 –707. 1

[Ellson et al., 2013] Ellson, J., Gansner, E., Hu, Y., Bilgin, A., and Perry, D. (2013). Graphviz | graphviz - graph visualization software. http://www.graphviz.org/. [last accessed:17.02.2013]. 5.2.7

[Ewert and Schultz, 1992] Ewert, R. A. and Schultz, S. M. (1992). Automatic hard disk bad sector remapping. 4.1.2.1

[Farrell, 2009] Farrell, P. F. J. (2009). A framework for automated digital forensic reporting. Master's thesis, Naval Postgraduate School Monterey. 2

[Federal Bureau of Investigation, 2011] Federal Bureau of Investigation (2011). FBI — International Cyber Ring That Infected Millions of Computers Dismantled. http://www.fbi.gov/news/stories/2011/november/malware_110911. [last accessed:17.02.2013]. 4.2.3.2

[Garfinkel, 2009] Garfinkel, S. (2009). Automating disk forensic processing with sleuthkit, xml and python. In Systematic Approaches to Digital Forensic Engineering, 2009. SADFE '09. Fourth International IEEE Workshop on, pages 73 –84. 2

[Garfinkel, 2010] Garfinkel, S. L. (2010). Digital forensics research: The next 10 years. Digital Investigation, 7, Supplement(0):S64 – S73. The Proceedings of the Tenth Annual DFRWS Conference. 1, 2

[Gephi Consortium, 2012] Gephi Consortium (2012). Gephi, an open source graph visualization and manipulation software. `https://gephi.org/`. [last accessed:17.02.2013]. 5.2.3

[Gruber, 2009] Gruber, T. (2009). Ontology. `http://tomgruber.org/writing/ontology-definition-2007.htm`. [last accessed:17.02.2013]. 5.1

[Harris, 2006] Harris, R. (2006). Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. Digital Investigation, 3, Supplement(0):44 – 49. <ce:title>The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06)</ce:title>. 4.2.3.3

[Hitzler et al., 2008a] Hitzler, P., Krötzsch, M., Rudolph, S., and Sure, Y. (2008a). Semantic Web, chapter 2, pages 15–32. Springer-Verlag Berlin Heidelberg. Struktur mit XML. 5.1.3

[Hitzler et al., 2008b] Hitzler, P., Krötzsch, M., Rudolph, S., and Sure, Y. (2008b). Semantic Web, chapter 3, pages 34–88. Springer-Verlag Berlin Heidelberg. Einfache Ontologien in RDF und RDF Schema. 5.1.3

[Hitzler et al., 2008c] Hitzler, P., Krötzsch, M., Rudolph, S., and Sure, Y. (2008c). Semantic Web, chapter 5, pages 123–159. Springer-Verlag Berlin Heidelberg. Ontologien in OWL. 5.1.3

[Huynh et al., 2003] Huynh, C., de Chazal, P., McErlean, D., Reilly, R., Hannigan, T., and Fleury, L. (2003). Automatic classification of shoeprints for use in forensic science based on the fourier transform. In Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on, volume 3, pages III – 569–72 vol.2. 1

[IADT Chicago, 2011] IADT Chicago (2011). Famous computer forensics case | iadt - chicago. `http://www.iadt.edu/Student-Life/IADT-Buzz/January-2011/Most-Famous-Case-Solved-Computer-Forensics`. [last accessed: 17.02.2013]. 4.1.1

[Kruse and Heiser, 2001] Kruse, W. and Heiser, J. (2001). Computer Forensics: Incident Response Essentials. Pearson Education. 4.1, 4.1.2, 4.1.2, 4.2.3.3

[Manson et al., 2007] Manson, D., Carlin, A., Ramos, S., Gyger, A., Kaufman, M., and Treichelt, J. (2007). Is the open way a better way? digital forensics using open source tools. In System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, page 266b. 4.3

[Microsoft, 2010] Microsoft (2010). Run and runonce registry keys (windows). http://msdn.microsoft.com/en-us/library/aa376977.aspx. [last accessed:17.02.2013]. 7.6.2

[National Institute of Justice (U.S.), 2001] National Institute of Justice (U.S.) (2001). Electronic crime scene investigation: a guide for first responders. NIJ guide. U.S. Dept. of Justice, Office of Justice Programs, National Institute of Justice. 1, 4.1.2

[National Institute of Justice (U.S.), 2004] National Institute of Justice (U.S.) (2004). Forensic examination of digital evidence: a guide for law enforcement. NIJ special report. U.S. Dept. of Justice, Office of Justice Programs, National Institute of Justice. 1, 4.1.2

[Neo Technology, Inc., 2006] Neo Technology, Inc. (2006). The neo database – a technology introduction. Technical report. 2, 5.3.1

[Neo Technology, Inc, 2013] Neo Technology, Inc (2013). Learn, develop, participate - neo4j: The world's leading graph database. http://www.neo4j.org/. [last accessed:17.02.2013]. 5.3.1, 5.3, 7.5

[Noy and McGuinness, 2001] Noy, N. F. and McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. Development, 32(1):1–25. 5.1, 5.1.1

[Oracle, 2012] Oracle (2012). Oracle vm virtualbox. https://www.virtualbox.org/. [last accessed:17.02.2013]. 8.1

[Pérez et al., 2009] Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of sparql. ACM Trans. Database Syst., 34(3):16:1–16:45. 5.1.5

[Prud'hommeaux, 2012] Prud'hommeaux, E. (2012). Sparql vs. sql - intro - cambridge semantics. http://www.cambridgesemantics.com/de/semantic-university/sparql-vs-sql-intro. [last accessed:17.02.2013]. 5.1.5

[Rutkowska, 2007] Rutkowska, J. (2007). Beyond the cpu: Defeating hardware based ram acquisition (part i: Amd case). In Black Hat 2007. 4.1.2.2

[Salzburg Research, 2012] Salzburg Research, A. (2012). Rdf-gravity. http://semweb.salzburgresearch.at/apps/rdf-gravity/. [last accessed:17.02.2013]. 5.2.4

[Schreck et al. Siemens CERT,] Schreck et al. Siemens CERT. Categorization of digital forensic data (random access memory). Not yet published. 4.2.2, 4.2.2

[Schuster, 2006] Schuster, A. (2006). Searching for processes and threads in microsoft windows memory dumps. Digit. Investig., 3:10–16. 4.3.2

[Sentinel Chicken Networks, 2010] Sentinel Chicken Networks (2010). RegLookup. `http://projects.sentinelchicken.org/reglookup/`. [last accessed:17.02.2013]. 4.3.3

[Stanford Center for Biomedical Informatics Research, 2013] Stanford Center for Biomedical Informatics Research (2013). The protégé ontology editor and knowledge acquisition system. `http://protege.stanford.edu/`. [last accessed:17.02.2013]. 5.2.2

[Tang and Daniels, 2005] Tang, Y. and Daniels, T. (2005). A simple framework for distributed forensics. In Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on, pages 163 – 169. 1

[The Associated Press, 2012] The Associated Press (2012). Computer disk may have cracked btk case - us news - crime & courts | nbc news. `http://www.nbcnews.com/id/6988048/`. [last accessed: 17.02.2013]. 4.1.1

[The GNOME Project, 2011] The GNOME Project (2011). dconf - gnome live! `https://live.gnome.org/dconf`. [last accessed:17.02.2013]. 4.2.3.1

[Tissieres and Oechslin, 2013] Tissieres, C. and Oechslin, P. (2013). ophcrack - browse /samdump2 at sourceforge.net. `http://sourceforge.net/projects/ophcrack/files/samdump2`. [last accessed: 17.02.2013]. 4.3.4

[Van Baar et al., 2008] Van Baar, R. B., Alink, W., and Van Ballegooij, A. R. (2008). Forensic memory analysis: Files mapped in memory. Digit. Investig., 5:S52–S57. 4.3.2

[Vidas, 2007] Vidas, T. (2007). The acquisition and analysis of random access memory. Journal of Digital Forensic Practice, 1(4):315–323. 4.1.2, 4.1.2.2

[Volatile Systems, 2012a] Volatile Systems (2012a). Commandreference21 - volatility - example usage cases and output for volatility 2.1 commands - an advanced memory forensics framework - google project hosting. `http://code.google.com/p/volatility/wiki/CommandReference21`. [last accessed:17.02.2013]. 4.3.2

[Volatile Systems, 2012b] Volatile Systems (2012b). The volatility framework: Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`. [last accessed:17.02.2013]. 4.3.2

[W3C, 2004] W3C (2004). Owl web ontology language reference. http://www.w3.org/TR/owl-ref. [last accessed:17.02.2013]. 5.1.3

[W3C, 2008] W3C (2008). Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query/. [last accessed:17.02.2013]. 5.1.5

[W3C, 2009a] W3C (2009a). Owl 2 web ontology language mapping to rdf graphs. http://www.w3.org/TR/2009/REC-owl2-mapping-to-rdf-20091027/. [last accessed:17.02.2013]. 5.1.3

[W3C, 2009b] W3C (2009b). Owl 2 web ontology language xml serialization. http://www.w3.org/TR/2009/REC-owl2-xml-serialization-20091027/. [last accessed:17.02.2013]. 5.1.3